MIỄN PHÍ 100% | Series tự học Flutter từ cơ bản tới nâng cao (cập nhật liên tục...)

Flutter là một bộ công cụ giao diện người dùng để tạo các ứng dụng nhanh, đẹp, được biên dịch nguyên bản cho thiết bị di động, web và máy tính để bàn với một ngôn ngữ lập trình và cơ sở code duy nhất.

Trong series bao gồm:

- Sách, video và tài liệu học git
- Lộ trình học Flutter từ cơ bản tới nâng cao
- Thực hành từng widget cơ bản và các vấn đề nâng cao với ví dụ chi tiết

👉 Giới thiệu mọi thứ về Cafedev tại đây

👉 Kho khoá học lập trình online chọn lọc(Cập nhật liên tục...)

Phần mở đầu	Sách, video và tài liệu
0.0	Kho sách Flutter
0.1	Nơi đăng ký nhận ebook lập trình, ebook công nghệ thông tin tại đây
0.2	Video học Flutter(Đang cập nhật...)

Phần 1	Giới thiệu
1.0	Giới thiệu chi tiết về Flutter và Series tự học Miễn Phí
1.1	Cài đặt Flutter với Android Studio
1.2	Tạo ứng dụng Flutter đầu tiên và giải thích chi tiết các trong project
1.3	Kiến trúc của Flutter

Phần mở đầu	Sách, video và tài liệu
1.4	Sự khác biệt giữa Flutter và React native
1.5	Giới thiệu lập trình với ngôn ngữ Dart

Phần 2	Kiến thức cơ bản về Flutter
2.0	Giới thiệu chi tiết về widgets và một số widgets cần phải biết trong Flutter
2.1	Tìm hiểu về bố cụ(layout) giao diện trong Flutter
2.2	Tự học Flutter | Tìm hiểu về Cử chỉ(Gestures) với giao diện trong Flutter
2.3	Tìm hiểu về cách quản lý state(trạng thái) trong Flutter
2.4	Tìm hiểu về các IDE dùng để code Flutter

Phần 3	Tìm hiểu về Widgets cần phải biết và hiểu
3.0	Tìm hiểu về widget Scaffold trong Flutter
3.1	Tìm hiểu về widget Container trong Flutter
3.2	Tìm hiểu về widget Row và Column trong Flutter
3.3	Tìm hiểu về widget Text trong Flutter
3.4	Tìm hiểu về widget Textfield trong Flutter
3.5	Tìm hiểu về widget button trong Flutter
3.6	Tìm hiểu về widget Stack trong Flutter

Phần mở đầu	Sách, video và tài liệu
3.7	Tìm hiểu về các widget về Forms trong Flutter
3.8	Tìm hiểu về widget Alert Dialogs trong Flutter
3.9	Tìm hiểu về widget Icon trong Flutter
3.10	Tìm hiểu về widget Image trong Flutter
3.11	Tìm hiểu về widget Card trong Flutter
3.12	Tìm hiểu về widget Tabbar trong Flutter
3.13	Tìm hiểu về widget Drawer trong Flutter
3.14	Tìm hiểu về widget list trong Flutter
3.15	Tìm hiểu về widget GridView trong Flutter
3.16	Tìm hiểu về widget Checkbox trong Flutter
3.17	Tìm hiểu về widget radio button trong Flutter
3.18	Tìm hiểu về widget Progress Bar trong Flutter
3.19	Tìm hiểu về widget Snackbar trong Flutter
3.20	Tìm hiểu về widget Tooltip trong Flutter
3.21	Tìm hiểu về widget Slider trong Flutter
3.22	Tìm hiểu về widget Switch trong Flutter
3.23	Tìm hiểu về widget Charts trong Flutter
3.24	Tìm hiểu về widget Bottom Navigation Bar trong Flutter
3.25	Tìm hiểu về widgets theme trong Flutter
3.26	Tìm hiểu về widget Table trong Flutter

Phần mở đầu	Sách, video và tài liệu
3.27	Tìm hiểu về animation trong Flutter

Phần 4	Phần nâng cao
4.0	Tìm hiểu về Navigation and Routing trong Flutter
4.1	Tìm hiểu về Android Platform-Specific Code trong từng nền tảng
4.2	Tìm hiểu về Flutter Packages
4.3	Tìm hiểu về Google Maps trong Flutter
4.4	Tìm hiểu về API REST trong Flutter
4.5	Tìm hiểu về Database trong Flutter
4.6	Tìm hiểu về Testing trong Flutter

Phần 5	So sánh
5.0	So sánh mọi thứ giữ Flutter vs Xamarin
5.1	So sánh mọi thứ giữ Flutter vs Kotlin
5.2	So sánh mọi thứ giữ Flutter vs Ionic

Phần 6	Phỏng vấn
6.0	Các câu hỏi phỏng vấn Flutter phần 1

Phần mở đầu	Sách, video và tài liệu
6.1	Các câu hỏi phỏng vấn Flutter phần 2
6.2	Các câu hỏi phỏng vấn Flutter phần 3

Các bài viết liên quan

Tự học Flutter | Giới thiệu lập trình với ngôn ngữ Dart

What is Dart Programming

Bởi **David Xuân** - 19 Tháng Ba, 2021 981 0

Dart là một ngôn ngữ lập trình hướng đối tượng mã nguồn mở, có mục đích chung với cú pháp kiểu C do **Google** phát triển **vào năm 2011** . Mục đích của lập trình Dart là tạo giao diện người dùng frontend cho web và ứng dụng dành cho thiết bị di động. Nó đang được phát triển tích cực, được biên dịch sang mã máy gốc để xây dựng ứng dụng di động, lấy cảm hứng từ các ngôn ngữ lập trình khác như Java, JavaScript, C # và Typed mạnh. Vì Dart là một ngôn ngữ biên dịch nên bạn không thể thực thi code của mình trực tiếp; thay vào đó, trình biên dịch phân tích cú pháp nó và chuyển nó thành code máy.

Nó hỗ trợ hầu hết các khái niệm chung của ngôn ngữ lập trình như lớp, giao diện, hàm, không giống như các ngôn ngữ lập trình khác. Ngôn ngữ Dart không hỗ trợ mảng trực tiếp. Nó hỗ trợ tập hợp, được sử dụng để sao chép cấu trúc dữ liệu như mảng, generic và kiểu tùy chọn.

Ví dụ sau đây cho thấy lập trình Dart đơn giản.

```
1  void main() {
2    for (int i = 0; i < 5; i++) {
```

```
      print('hello ${i + 1}');
    }
  }
```

Nội dung chính

1. Kiểu dữ liệu

Dart là một ngôn ngữ lập trình Strongly Typed. Nó có nghĩa là, mỗi giá trị bạn sử dụng trong ngôn ngữ lập trình của mình có một kiểu là chuỗi hoặc số và phải được biết chính xác là kiểu dữ liệu gì trước khi code được biên dịch. Ở đây, chúng ta sẽ thảo luận về các kiểu dữ liệu cơ bản phổ biến nhất được sử dụng trong ngôn ngữ lập trình Dart.

Loại dữ liệu	**Thí dụ**	**Mô tả**
Chuỗi	String myName = cafedev.vn;	Nó chứa văn bản. Trong phần này, bạn có thể sử dụng dấu ngoặc kép đơn hoặc kép. Khi bạn quyết định dấu ngoặc kép, bạn phải nhất quán với lựa chọn của mình.
num, int, double	int age = 25;double = 125,50;	Kiểu dữ liệu num là viết tắt của một số. Dart có hai loại số:Số nguyên (Là một số không có chữ số thập phân.)Double (Nó là một số có chữ số thập phân.)
Boolean	bool var_name = true;Hoặcbool var_name = false;	Nó sử dụng từ khóa bool để biểu thị giá trị Boolean true và false.
vật	Person = Persion()	Nói chung, mọi thứ trong Dart là một đối tượng (ví dụ: Số nguyên, Chuỗi). Nhưng một đối tượng cũng có thể phức tạp hơn.

2. Các biến và hàm

Các biến là không gian tên trong bộ nhớ lưu trữ các giá trị. Tên của một biến được gọi là định danh(**identifiers**). Chúng là nơi chứa dữ liệu, có thể lưu trữ giá trị của bất kỳ kiểu nào. Ví dụ:

1. var myAge = 50 ;

Ở đây, **myAge** là một biến lưu trữ giá trị số nguyên 50. Chúng ta cũng có thể cho nó là int và double. Tuy nhiên, Dart có một tính năng Type Inference, suy luận các loại giá trị. Vì vậy, nếu bạn tạo một biến với từ khóa **var** , Dart có thể suy ra biến đó thuộc loại số nguyên.

Bên cạnh biến, Hàm là một tính năng cốt lõi khác của bất kỳ ngôn ngữ lập trình nào. Hàm là một tập hợp các câu lệnh thực hiện một nhiệm vụ cụ thể. Chúng được tổ chức thành các khối mã logic có thể đọc được, có thể bảo trì và có thể tái sử dụng. Khai báo hàm chứa tên hàm, kiểu trả về và các tham số. Ví dụ sau giải thích hàm được sử dụng trong lập trình Dart.

```
1  //Function declaration
2  num addNumbers(num a, num b) {
3      // Here, we use num as a type because it should work with int an
4      return a + b;
5  }
6  var price1 = 29.99;
7  var price2 = 20.81;
8  var total = addNumbers(price1, price2);
9  var num1 = 10;
10 var num2 = 45;
11 var total2 = addNumbers(num1, num2);
```

3. Operators

Ngôn ngữ Dart hỗ trợ tất cả các toán tử, vì bạn đã quen thuộc với các ngôn ngữ lập trình khác như C, Kotlin và Swift. Tên của Operators

được liệt kê dưới đây:

- Số học
- Bằng
- Tăng và giảm
- Logical

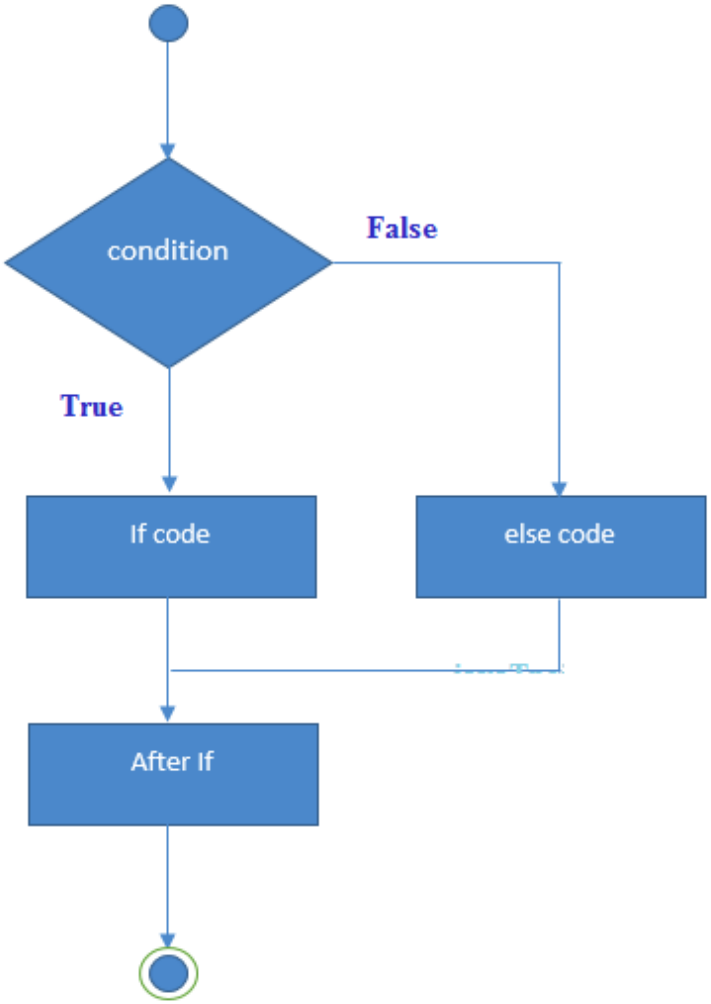
- So sánh

4. Ra quyết định và các vòng lặp

Ra quyết định là một tính năng cho phép bạn đánh giá một điều kiện trước khi các câu lệnh được thực thi. Ngôn ngữ Dart hỗ trợ các loại câu lệnh ra quyết định sau:

- Câu lệnh if
- Câu lệnh if-else
- Chuyển đổi câu lệnh

Sơ đồ dưới đây giải thích rõ ràng hơn.



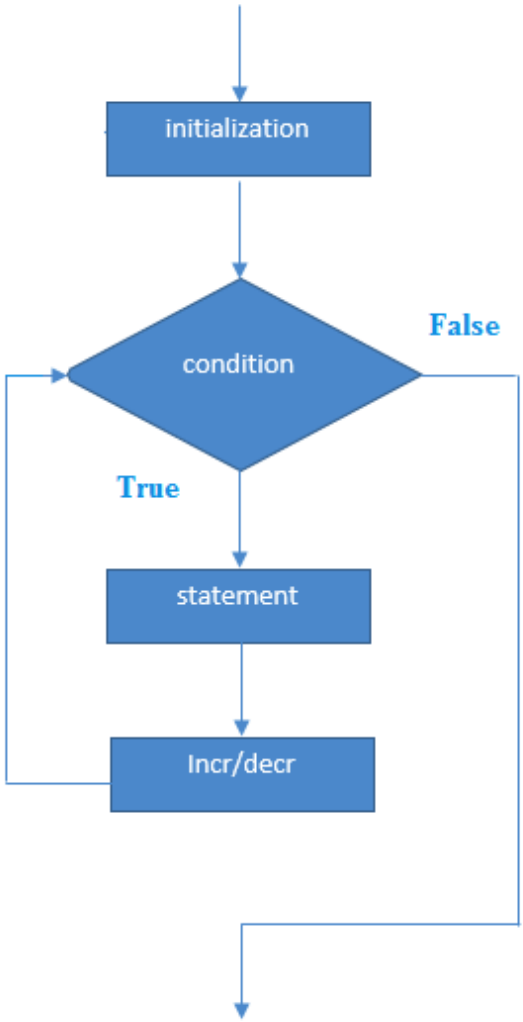
Thí dụ

```
void main() {
   var num = 12;
   if (num % 2 = = 0) {
      print("Number is Even.");
   } else {
      print("Number is Odd.");
   }
}
```

Các vòng lặp được sử dụng để thực thi một khối code lặp đi lặp lại cho đến khi một điều kiện được chỉ định trở thành đúng. Ngôn ngữ Dart hỗ trợ các loại câu lệnh lặp sau:

- for
- for..in
- while
- do..while

Sơ đồ dưới đây giải thích rõ ràng hơn.



Thí dụ

```
void main() {
   var name = ["Peter", "Rinky Ponting", "Abhishek"];

   for (var prop in name) {
      print(prop);
   }
}
```

5. Bình luận(Comments)

Nhận xét là những dòng code **không thực thi được** . Chúng là một trong những khía cạnh chính của tất cả các ngôn ngữ lập trình. Mục đích của việc này là cung cấp thông tin về dự án, biến hoặc một hoạt động. Có ba loại nhận xét trong lập trình Dart:

- **Thực hiện các nhận xét định dạng:** Đó là một nhận xét dòng đơn (///)
- **Khối nhận xét:** Đó là một nhận xét nhiều dòng (/*...*/)
- **Nhận xét Tài liệu:** Đây là nhận xét tài liệu được sử dụng để mô tả cho thành viên và các kiểu (///)

6. Tiếp tục và Phá vỡ(Continue and Break)

Dart cũng đã sử dụng từ khóa Continue và Break trong vòng lặp, và ở những nơi khác nó được yêu cầu. Câu lệnh continue cho phép bạn bỏ qua đoạn code còn lại bên trong vòng lặp và ngay lập tức chuyển đến bước lặp tiếp theo của vòng lặp. Chúng ta có thể hiểu nó từ ví dụ sau.

Thí dụ

```
void main() {
  for(int i=1;i<=10;i++){
    if(i==5){
      print("Hello");
      continue; //it will skip the rest statement
    }
    print(i);
  }
}
```

Câu lệnh **break** cho phép bạn kết thúc hoặc dừng dòng hiện tại của một chương trình và tiếp tục thực hiện sau phần thân của vòng lặp. Ví dụ sau đây sẽ giải thích chi tiết.

Thí dụ

```
void main() {
  for(int i=1;i<=10;i++){
    if(i==5){
      print("Hello");
      break;//it will terminate the rest statement
    }
    print(i);
  }
}
```

7. Từ khóa Final và Const

Chúng ta có thể sử dụng một từ khóa Final để hạn chế người dùng. Nó có thể được áp dụng trong nhiều ngữ cảnh, chẳng hạn như biến, lớp và phương thức.

Từ khóa Const dùng để khai báo hằng. Chúng ta không thể thay đổi giá trị của từ khóa const sau khi gán nó.

Thí dụ

```
void main() {
  final a = 100;
  const pi = 3.14;
  print(a);
  print(pi);
}
```

8. Lập trình hướng đối tượng

Dart là một ngôn ngữ lập trình hướng đối tượng, có nghĩa là mọi giá trị trong Dart đều là một đối tượng. Một số cũng là một đối tượng trong ngôn ngữ Dart. Lập trình Dart hỗ trợ khái niệm về các tính năng OOP như đối tượng, lớp, giao diện, v.v.

Đối tượng: Đối tượng là một thực thể, có trạng thái và hành vi. Nó có thể là vật lý hoặc logic. Trong Dart, mọi giá trị đều là một đối tượng, ngay cả những giá trị nguyên thủy như văn bản và số. Dart cũng có thể cho phép bạn xây dựng đối tượng tùy chỉnh của mình để thể hiện các mối quan hệ phức tạp hơn giữa các dữ liệu.

Lớp: Một lớp là một tập hợp các đối tượng. Nó có nghĩa là các đối tượng được tạo ra với sự trợ giúp của các lớp vì mọi đối tượng đều cần một bản thiết kế dựa trên đó bạn có thể tạo một đối tượng riêng lẻ. Một định nghĩa lớp bao gồm những điều sau:

- Thuộc tính
- Phương pháp
- Constructor
- Getters và setters

Hãy để chúng ta xem một ví dụ, giúp bạn hiểu khái niệm OOP một cách dễ dàng.

```
class Mobile {
  // Property Declaration
  String color, brandName, modelName;

  // Method Creation
  String calling() {
    return "Mobile can do call to everyone.";
  }
  String musicPlay() {
    return "Mobile can play all types of Music.";
  }
  String clickPicture() {
    return "Mobile can take pictures.";
  }
}

void main() {
  // Object Creation
  var myMob = new Mobile();

  // Accessing Class's Property
  myMob.color = "Black";
  myMob.brandName = "Apple Inc.";
  myMob.modelName = "iPhone 11 Pro";

  //Display Output
  print(myMob.color);
  print(myMob.modelName);
  print(myMob.brandName);
  print(myMob.calling());
  print(myMob.musicPlay());
  print(myMob.clickPicture());
}
```

Trong ví dụ trên, chúng ta định nghĩa một lớp **Mobile** , có ba biến kiểu chuỗi và ba hàm hoặc phương thức. Sau đó, chúng ta tạo một hàm **main** mà Dart sẽ thực thi đầu tiên khi ứng dụng của bạn khởi động. Bên trong main, chúng ta tạo một **đối tượng** để truy cập các thuộc tính của lớp. Cuối cùng, chúng ta in đầu ra.

Tài liệu từ cafedev:

- Full series tự học Flutter từ cơ bản tới nâng cao tại đây nha.
- **Các nguồn kiến thức MIỄN PHÍ VÔ GIÁ từ cafedev tại đây**

Nếu bạn thấy hay và hữu ích, bạn có thể tham gia các kênh sau của cafedev để nhận được nhiều hơn nữa:

- **Group Facebook**
- **Fanpage**
- **Youtube**
- **Instagram**
- **Twitter**
- **Linkedin**
- **Pinterest**
- **Trang chủ**

Chào thân ái và quyết thắng!

Đăng ký kênh youtube để ủng hộ Cafedev nha các bạn, Thanks you!

Bài trước

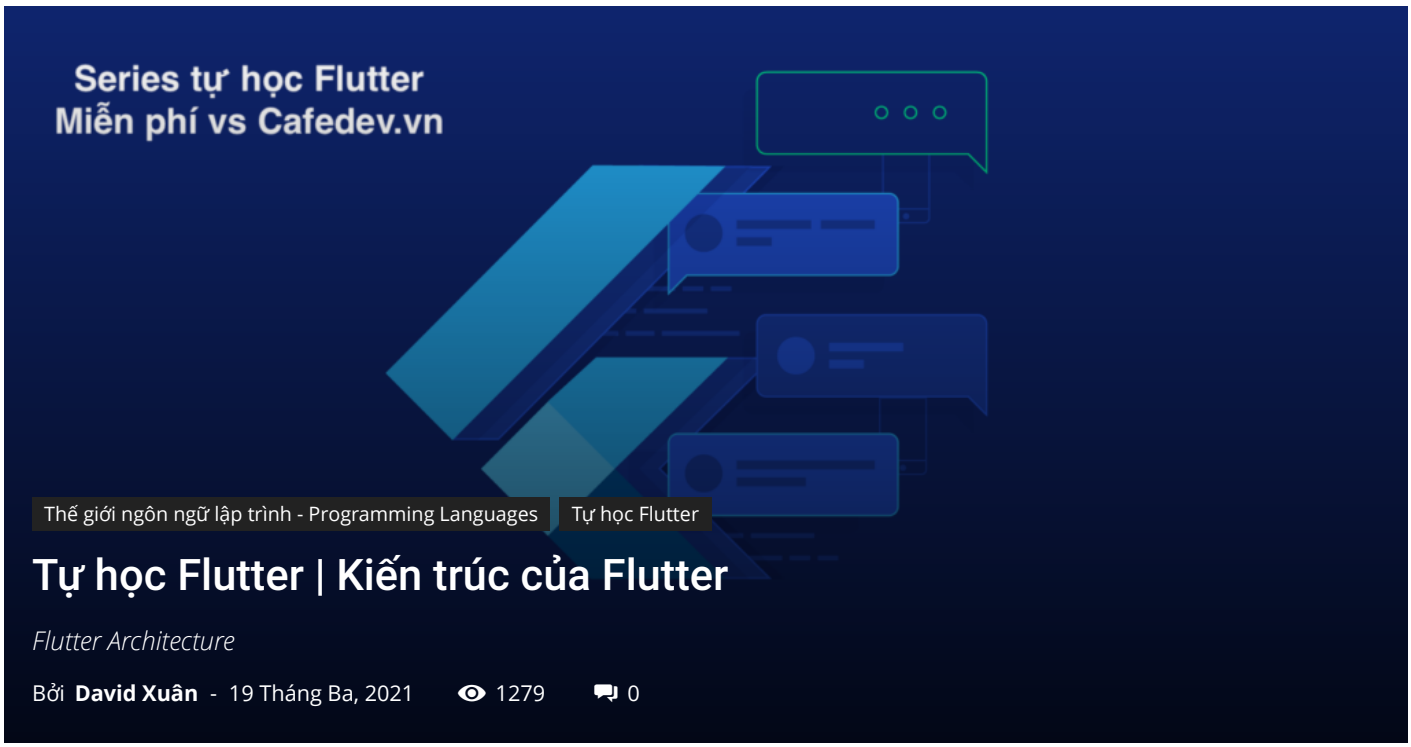
Tự học Flutter | Sự khác biệt giữa Flutter và React native

Bài tiếp theo

Tự học Flutter | Giới thiệu chi tiết về widgets và một số widgets cần phải biết trong Flutter

David Xuân

https://cafedev.vn/



Nội dung chính

1. Kiến trúc Flutter

Trong phần này, chúng ta sẽ thảo luận về kiến trúc của Flutter framework. Kiến trúc Flutter chủ yếu bao gồm bốn thành phần.

1. Động cơ Flutter(Flutter Engine)
2. Thư viện nền tảng(Foundation Library)
3. Vật dụng(Widgets)
4. Thiết kế các widget cụ thể(Design Specific Widgets)

2. Flutter Engine

Nó là một cổng để giúp chạy các ứng dụng di động chất lượng cao và cơ bản dựa trên ngôn ngữ C ++. Nó triển khai các thư viện lõi Flutter bao gồm animation và đồ họa, tệp và mạng I / O, kiến trúc plugin, hỗ trợ trợ năng và thời gian chạy dart để phát triển, biên dịch và chạy các ứng dụng Flutter. Phải sử dụng thư viện đồ họa mã nguồn mở của Google, **Skia** , để hiển thị đồ họa cấp thấp.

3. Thư viện nền tảng(Foundation Library)

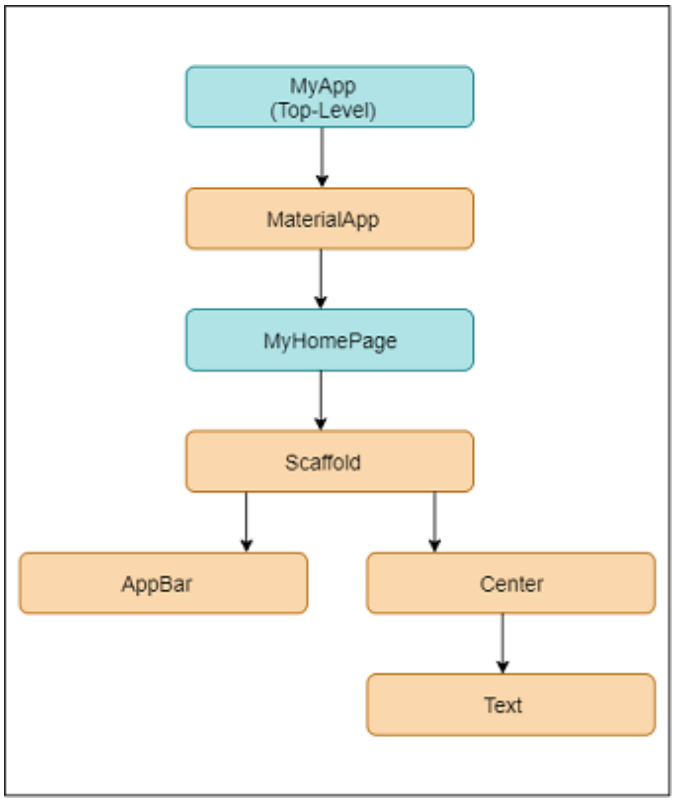
Nó chứa tất cả các gói cần thiết cho các khối build cơ bản để viết một ứng dụng Flutter. Các thư viện này được viết bằng ngôn ngữ Dart.

4. Vật dụng(widget)

Trong Flutter, mọi thứ đều là một widget, đó là khái niệm cốt lõi của framework. Widget trong Flutter về cơ bản là một thành phần giao diện người dùng ảnh hưởng và kiểm soát chế độ xem và giao diện của ứng dụng. Nó đại diện cho một mô tả bất biến về một phần của giao diện người dùng và bao gồm đồ họa, văn bản, hình dạng và animation được tạo bằng các widget. Các widget tương tự như các thành phần React.

Trong Flutter, ứng dụng tự nó là một widget chứa nhiều widget con. Điều đó có nghĩa là ứng dụng là tiện ích con cấp cao nhất và giao diện người dùng của nó được xây dựng bằng cách sử dụng một hoặc nhiều tiện ích con, bao gồm các tiện ích con phụ. Tính năng này giúp bạn tạo một giao diện người dùng phức tạp rất dễ dàng.

Chúng ta có thể hiểu nó từ ví dụ hello world đã tạo ở phần trước. Ở đây, chúng ta sẽ giải thích ví dụ bằng sơ đồ sau.



Trong ví dụ trên, chúng ta có thể thấy rằng tất cả các thành phần đều là các widget có chứa các widget con. Do đó, ứng dụng Flutter tự nó là một widget.

5. Thiết kế các widget cụ thể

Framework Flutter có hai bộ widget phù hợp với các ngôn ngữ thiết kế cụ thể. Đây là Material Design cho ứng dụng Android và Cupertino Style cho ứng dụng IOS.

6. Cử chỉ(Gestures)

Nó là một tiện ích cung cấp sự tương tác (cách lắng nghe và phản hồi) trong Flutter bằng cách sử dụng GestureDetector. **GestureDector** là một widget vô hình, bao gồm tương tác chạm, kéo và mở rộng quy mô của widget con của nó. Chúng ta cũng có thể sử dụng các tính năng tương tác khác vào các widget hiện có bằng cách soạn thảo với widget GestureDetector.

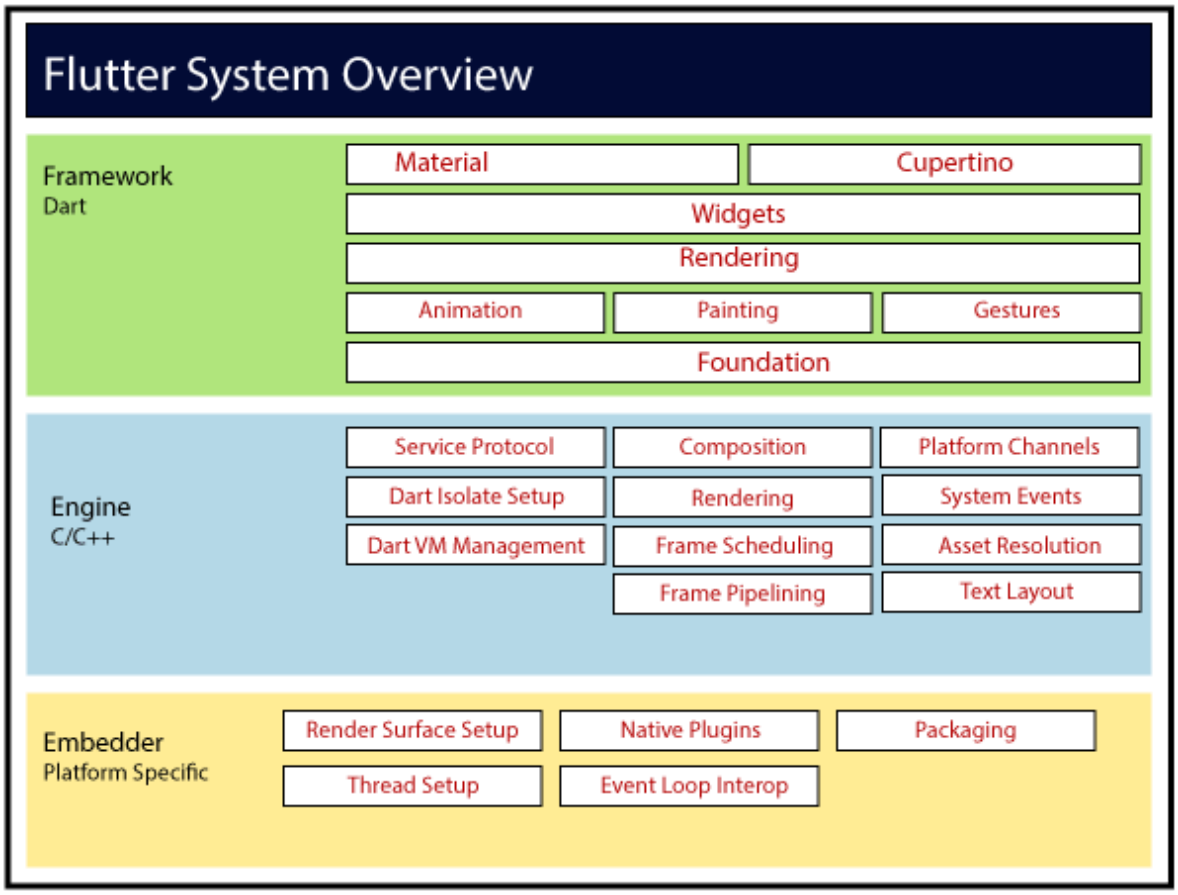
7. Quản lý State

Tiện ích Flutter duy trì trạng thái của nó bằng cách sử dụng một tiện ích đặc biệt, StatefulWidget. Nó luôn được tự động hiển thị lại bất cứ khi nào trạng thái bên trong của nó bị thay đổi. Kết xuất được tối ưu hóa bằng cách tính toán khoảng cách giữa giao

diện người dùng tiện ích con cũ và mới và chỉ hiển thị những thứ cần thiết là các thay đổi.

8. Lớp(Layers)

Layers là một khái niệm quan trọng của khung Flutter, được nhóm thành nhiều loại về mức độ phức tạp và được sắp xếp theo cách tiếp cận từ trên xuống. Lớp trên cùng là giao diện người dùng của ứng dụng, dành riêng cho nền tảng Android và iOS. Lớp trên cùng thứ hai chứa tất cả các widget gốc của Flutter. Lớp tiếp theo là lớp kết xuất, lớp này hiển thị mọi thứ trong ứng dụng Flutter. Sau đó, các lớp đi xuống Gestures, foundation library, engine, và cuối cùng là mã cốt lõi dành riêng cho nền tảng. Sơ đồ sau chỉ định các lớp trong phát triển ứng dụng Flutter.



Cài ứng dụng cafedev để dễ dàng cập nhật tin và học lập trình mọi lúc mọi nơi tại đây.

Tài liệu từ cafedev:

- Full series tự học Flutter từ cơ bản tới nâng cao tại đây nha.
- **Các nguồn kiến thức MIỄN PHÍ VÔ GIÁ từ cafedev tại đây**

Nếu bạn thấy hay và hữu ích, bạn có thể tham gia các kênh sau của cafedev để nhận được nhiều hơn nữa:

- **Group Facebook**
- **Fanpage**
- **Youtube**
- **Instagram**
- **Twitter**
- **Linkedin**
- **Pinterest**
- **Trang chủ**

Chào thân ái và quyết thắng!

Đăng ký kênh youtube để ủng hộ Cafedev nha các bạn, Thanks you!

Bài trước

Tự học Flutter | Tạo ứng dụng Flutter đầu tiên và giải thích chi tiết các trong project

Bài tiếp theo

Tự học Flutter | Sự khác biệt giữa Flutter và React native

David Xuân

https://cafedev.vn/

Tự học Flutter | Tìm hiểu về Testing trong Flutter

Flutter Testing

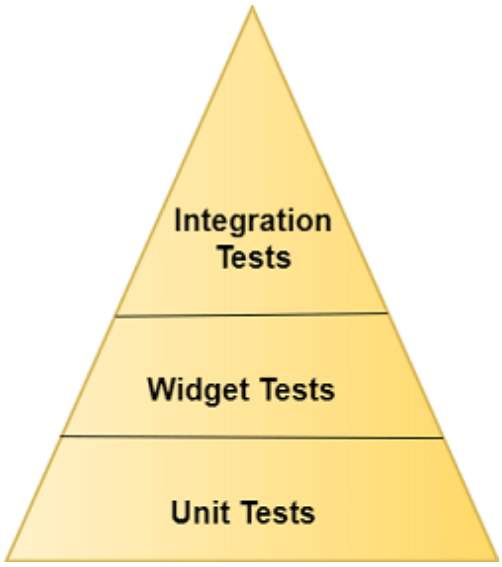
Bởi **David Xuân** - 3 Tháng Năm, 2021 566 0

Kiểm tra(Testing) là một hoạt động được sử dụng để xác minh và xác thực một phần mềm hoặc ứng dụng không có lỗi và đáp ứng các yêu cầu của người dùng. Nó đảm bảo rằng kết quả thực tế phù hợp với kết quả mong đợi. Nó cũng giúp cải thiện phần mềm hoặc ứng dụng về hiệu quả, khả năng sử dụng và độ chính xác.

Kiểm tra là một trong những giai đoạn quan trọng nhất trong vòng đời phát triển ứng dụng để đảm bảo ứng dụng có chất lượng cao. Đây là giai đoạn tiêu tốn nhiều nhất trong quá trình phát triển ứng dụng hoặc phần mềm.

Framework Flutter cung cấp hỗ trợ tuyệt vời cho việc **kiểm tra tự động** một ứng dụng. Nói chung, kiểm thử tự động phân loại thành ba loại để kiểm tra hoàn toàn một ứng dụng. Chúng như sau:

1. Kiểm tra đơn vị(Unit Testing)
2. Kiểm tra widget(Widget Testing)
3. Tích hợp Thử nghiệm(Integration testing)



Nội dung chính

1. Kiểm tra đơn vị

Đây là Phương thức dễ nhất để kiểm tra một ứng dụng hoặc phần mềm. Nó kiểm tra một hàm, phương thức hoặc lớp. Mục tiêu của kiểm thử đơn vị là đảm bảo tính đúng đắn của mã trong nhiều điều kiện khác nhau. Nói chung, kiểm thử đơn vị không tương tác với đầu vào của người dùng, hiển thị trên màn hình, đọc hoặc ghi dữ liệu từ đĩa và không sử dụng các phần phụ thuộc bên ngoài theo mặc định. Khi bạn sử dụng các phần phụ thuộc bên ngoài, chúng sẽ bị chế nhạo với các gói như Mockito.

2. Kiểm tra widget

Kiểm tra widget được sử dụng để kiểm tra một widget. Mục tiêu của thử nghiệm này là để đảm bảo rằng giao diện người dùng của widget trông và tương tác với các widget khác như mong đợi. Quá trình kiểm tra widget tương tự như kiểm thử đơn vị, nhưng nó toàn diện hơn kiểm thử đơn vị. Thử nghiệm này liên quan đến nhiều lớp và yêu cầu một môi trường thử nghiệm để tìm ra nhiều lỗi hơn. Một widget, đang được thử nghiệm, có thể nhận và phản hồi các hành động và sự kiện của người dùng và có thể khởi tạo các widget con.

3. Tích hợp Thử nghiệm

Kiểm thử tích hợp bao gồm cả kiểm tra ở trên cùng với các thành phần bên ngoài của ứng dụng. Nó xác thực một ứng dụng hoàn chỉnh hoặc một phần lớn của ứng dụng. Mục

đích của kiểm tra tích hợp là để đảm bảo rằng tất cả các widget và dịch vụ hoạt động cùng nhau như mong đợi. Nó cũng có thể sử dụng để xác minh hiệu suất của ứng dụng. Nói chung, thử nghiệm tích hợp chạy trên các thiết bị thực như trình giả lập Android hoặc trình mô phỏng iOS.

Sự cân bằng giữa các loại thử nghiệm khác nhau được đưa ra dưới đây:

	Kiểm tra đơn vị	**Kiểm tra widget**	**Tích hợp Thử nghiệm**
Sự tự tin	Thấp	Cao hơn	Cao nhất
Bảo trì	Tốn phí	Thấp cao hơn	Cao nhất
Sự phụ thuộc ở các dữ án	Vài cái	Hơn	Phần lớn
Tốc độ thực thi	Nhanh chóng	Nhanh chóng	Chậm

Chúng tôi biết rằng, trong Flutter, mọi thứ đều là một widget. Vì vậy, ở đây, chúng ta sẽ thảo luận chi tiết về việc kiểm tra widget.

4. Giới thiệu về Kiểm tra widget

Trong kiểm tra widget, bạn cần một số công cụ bổ sung được cung cấp bởi gói **flut_test** . Gói này cung cấp các công cụ sau để kiểm tra widget.

- **WidgetTester:** Nó cho phép xây dựng và tương tác với các widget trong môi trường thử nghiệm.
- **testWidgets():** Phương thức này tự động tạo một WidgetTester cho mỗi trường hợp thử nghiệm. Nó được sử dụng như một hàm test () bình thường. Nó chấp nhận hai đối số: mô tả thử nghiệm và mã thử nghiệm.
- **Lớp Finder:** Nó được sử dụng để tìm kiếm các widget trong môi trường thử nghiệm.
- **Lớp Matcher:** Nó giúp xác minh xem một lớp Finder định vị một widget hay nhiều widget trong môi trường thử nghiệm.

Hãy để chúng ta tìm hiểu cách tất cả những điều trên phù hợp với nhau bằng các bước sau:

Bước 1: Thêm sự phụ thuộc của Flagship_test.

Trong bước đầu tiên, chúng ta cần thêm một phần phụ thuộc flay_test trong file **pubspec.yaml** . Theo mặc định, nó đã được thêm vào phần phụ thuộc.

Bước 2: Tạo một widget để kiểm tra.

Tiếp theo, chúng ta phải tạo một widget để thực hiện thử nghiệm. Đoạn mã dưới đây tạo một widget chứa tiêu đề và thông báo để hiển thị trên màn hình.

```
class MyAppWidget extends StatelessWidget {
  final String title;
  final String message;

  const MyWidget({
    Key key,
    @required this.title,
    @required this.message,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Testing Demo',
      home: Scaffold(
        appBar: AppBar(
          title: Text(title),
        ),
        body: Center(
          child: Text(message),
        ),
      ),
    );
  }
}
```

Bước 3: Tạo thử nghiệm testWidgets.

Để kiểm tra widget, hãy sử dụng phương thức **testWidget()** . Phương thức này cho phép chúng ta xác định một thử nghiệm và chấp nhận hai đối số: mô tả thử nghiệm và mã thử nghiệm. Nó cũng tạo ra một **WidgetTester** để làm việc với widget. Đoạn mã sau xác minh rằng MyAppWidget hiển thị tiêu đề và tin nhắn.

```
void main() {
  testWidgets(' The widget contains a title and message', (WidgetTes
    // Write test code here.
  });
}
```

Bước 4: Xây dựng widget bằng WidgetTester.

WidgetTester cung cấp phương thức **pumpWidget ()** để xây dựng và hiển thị widget được cung cấp. Nó tạo ra bản sao của MyAppWidget hiển thị 'Ti' và 'Msg' làm tiêu đề và thông điệp tương ứng. Đoạn mã sau đây giải thích rõ ràng hơn.

```
void main() {
    testWidgets(The widget contains a title and message', (WidgetTes
        await tester.pumpWidget(MyWidget(title: 'Ti', message: 'Msg'

  });
}
```

Bước 5: Tìm kiếm widget bằng Finder.

Trong bước này, chúng ta sẽ sử dụng lớp Finder để tìm kiếm tiêu đề và tin nhắn trên cây widget. Nó cho phép chúng tôi xác minh rằng widget được hiển thị chính xác. Để làm điều này, chúng ta cần sử dụng phương thức **find.text ()** .

```
void main() {
    testWidgets(The widget contains a title and message', (WidgetTes
    await tester.pumpWidget(MyWidget(title: 'Ti', message: 'Msg'));

    final titleFinder = find.text('Ti');
    final messageFinder = find.text('Msg');

  });
}
```

Bước 6: Xác minh widget bằng Matcher.

Cuối cùng, chúng ta cần xác minh tin nhắn văn bản xuất hiện trên màn hình bằng cách sử dụng lớp Matcher. Nó đảm bảo rằng widget xuất hiện trên màn hình chính xác một lần. Chúng ta có thể xem đoạn mã sau để hiểu nó.

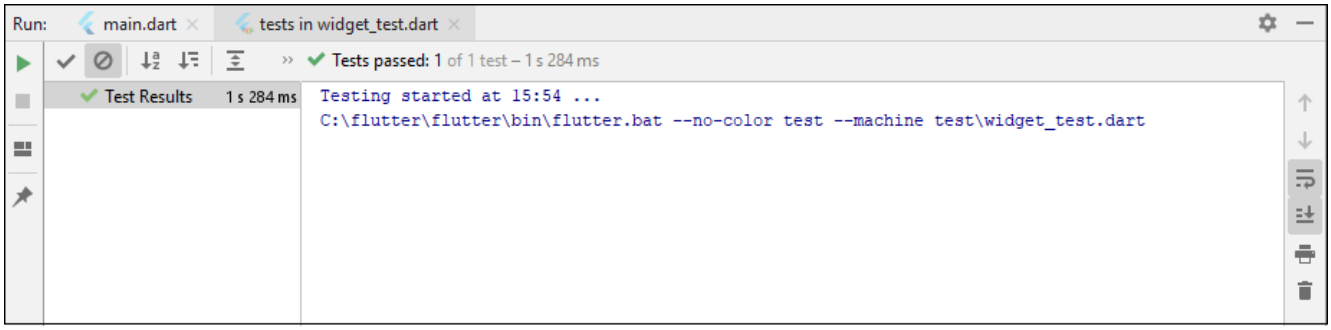
```
void main() {
    testWidgets(The widget contains a title and message', (WidgetTes
    await tester.pumpWidget(MyWidget(title: 'Ti', message: 'Msg'));

    final titleFinder = find.text('Ti');
    final messageFinder = find.text('Msg');

    expect(titleFinder, findsOneWidget);
    expect(messageFinder, findsOneWidget);
  });
}
```

Bây giờ, chúng ta sẽ xem ví dụ làm việc để hiểu khái niệm kiểm tra widget. Đầu tiên, tạo một dự án trong Android Studio và điều hướng đến thư mục thử nghiệm của thư mục dự án. Bây giờ, hãy mở tệp **widget_test.dart** và thay thế mã sau:

```
import 'package:flutter/material.dart';
import 'package:flutter_test/flutter_test.dart';

void main() {
    testWidgets(The widget contains a title and message', (WidgetTes
    // Create the widget by telling the tester to build it.
    await tester.pumpWidget(MyWidget(title: 'Ti', message: 'Msg'));

    // Create the Finders.
    final titleFinder = find.text('Ti');
    final messageFinder = find.text('Msg');

    expect(titleFinder, findsOneWidget);
    expect(messageFinder, findsOneWidget);
  });
}

class MyAppWidget extends StatelessWidget {
  final String title;
  final String message;
```

```
    const MyAppWidget({
      Key key,
      @required this.title,
      @required this.message,
    }) : super(key: key);

    @override
    Widget build(BuildContext context) {
      return MaterialApp(
        title: 'Flutter Testing Demo',
        home: Scaffold(
          appBar: AppBar(
            title: Text(title),
          ),
          body: Center(
            child: Text(message),
          ),
        ),
      );
    }
  }
```

Để thực hiện kiểm tra, hãy chuyển đến menu **Chạy** và chọn tùy chọn "kiểm tra trong widget_test.dart". Nó sẽ chạy thử nghiệm và cho kết quả như màn hình sau:



Cài ứng dụng cafedev để dễ dàng cập nhật tin và học lập trình mọi lúc mọi nơi tại đây.

Tài liệu từ cafedev:

- Full series tự học Flutter từ cơ bản tới nâng cao tại đây nha.
- **Các nguồn kiến thức MIỄN PHÍ VÔ GIÁ từ cafedev tại đây**

Nếu bạn thấy hay và hữu ích, bạn có thể tham gia các kênh sau của cafedev để nhận được nhiều hơn nữa:

- **Group Facebook**
- **Fanpage**
- **Youtube**
- **Instagram**
- **Twitter**
- **Linkedin**
- **Pinterest**
- **Trang chủ**

Chào thân ái và quyết thắng!

Đăng ký kênh youtube để ủng hộ Cafedev nha các bạn, Thanks you!

Bài trước

Tự học Flutter | Tìm hiểu về Database trong Flutter

Bài tiếp theo

Tự học Flutter | So sánh mọi thứ giữ Flutter vs Xamarin

David Xuân

https://cafedev.vn/

Thế giới ngôn ngữ lập trình - Programming Languages | Tự học Flutter

Tự học Flutter | Tìm hiểu về bố cụ(layout) giao diện trong Flutter

Flutter Layouts

Bởi **David Xuân** - 29 Tháng Ba, 2021 1466 0

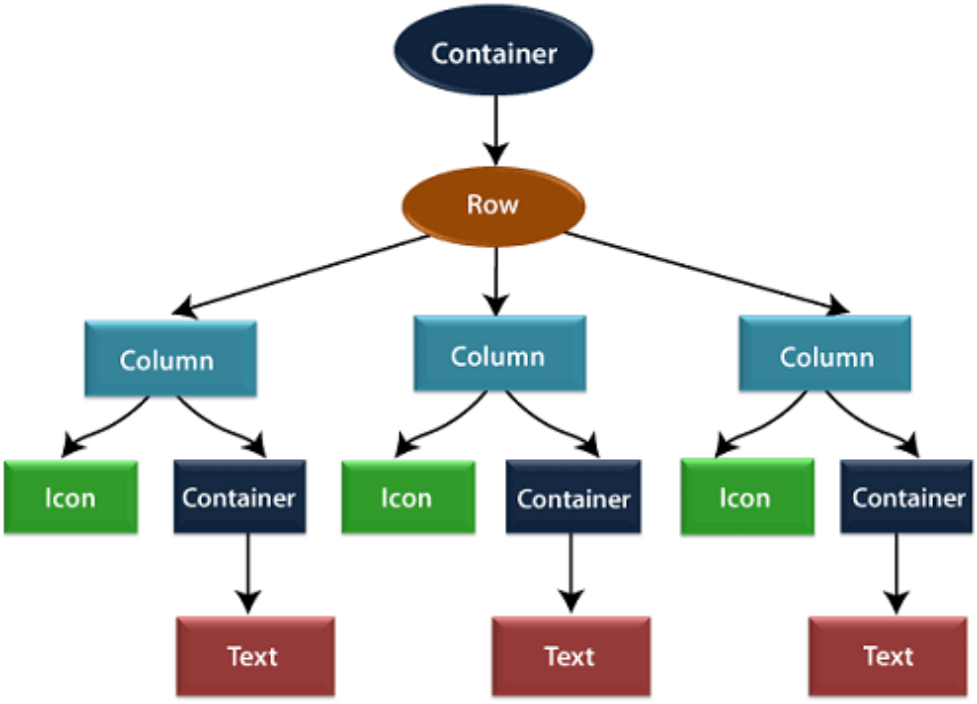
Nội dung chính

1. Bố cục Flutter

Khái niệm chính của cơ chế bố trí là widget. Chúng ta biết rằng sự flutter giả định mọi thứ như một widget Vì vậy, hình ảnh, biểu tượng, văn bản và thậm chí cả bố cục(layout) của ứng dụng của bạn đều là widget. Ở đây, một số thứ bạn không thấy trên giao diện người dùng ứng dụng của mình, chẳng hạn như các hàng, cột và lưới sắp xếp, ràng buộc và căn chỉnh các widget hiển thị cũng là các widget.

Flutter cho phép chúng ta tạo bố cục bằng cách soạn nhiều widget để xây dựng các widget phức tạp hơn. **Ví dụ** , chúng ta có thể thấy hình ảnh dưới đây hiển thị ba biểu tượng với nhãn bên dưới mỗi biểu tượng.

Trong hình ảnh thứ hai, chúng ta có thể thấy bố cục trực quan của hình ảnh trên. Hình ảnh này hiển thị một hàng gồm ba cột và các cột này chứa một biểu tượng và nhãn.



Trong hình trên, vùng **chứa** là một lớp widget cho phép chúng ta tùy chỉnh widget con. Nó chủ yếu được sử dụng để thêm đường viền, đệm, lề, màu nền và nhiều thứ khác. Tại đây, widget văn bản nằm dưới vùng chứa để thêm lề. Toàn bộ hàng cũng được đặt trong một vùng chứa để thêm lề và phần đệm xung quanh hàng. Ngoài ra, phần còn lại của giao diện người dùng được kiểm soát bởi các thuộc tính như màu sắc, kiểu văn bản, v.v.

2. Bố trí một widget

Hãy để chúng ta tìm hiểu cách chúng ta có thể tạo và hiển thị một widget đơn giản. Các bước sau đây cho biết cách bố trí widget:

Bước 1: Đầu tiên, bạn cần chọn một Bố cục widget.

Bước 2: Tiếp theo, tạo một widget hiển thị.

Bước 3: Sau đó, thêm widget hiển thị vào widget layout.

Bước 4: Cuối cùng, thêm widget bố cục vào trang mà bạn muốn hiển thị.

3. Các loại widget bố cục

Chúng tôi có thể phân loại widget bố cục thành hai loại:

1. widget đơn
2. widget đa

3.1 Các widget đơn

widget bố cục con duy nhất là một loại widget, có thể chỉ có **một widget** bên trong widget bố cục mẹ. Các widget này cũng có thể chứa chức năng bố cục đặc biệt. Flutter cung cấp cho chúng ta nhiều widget con để làm cho giao diện người dùng của ứng dụng trở nên hấp dẫn. Nếu chúng ta sử dụng các widget này một cách thích hợp, nó có thể tiết kiệm thời gian của chúng ta và làm cho code ứng dụng dễ đọc hơn. Danh sách các loại widget đơn lẻ khác nhau là:

Container: Đây là widget bố cục phổ biến nhất cung cấp các tùy chọn có thể tùy chỉnh để đặt màu, định vị và định cỡ các widget.

```
Center(
  child: Container(
    margin: const EdgeInsets.all(15.0),
    color: Colors.blue,
    width: 42.0,
    height: 42.0,
  ),
)
```

Padding: Nó là một widget được sử dụng để sắp xếp widget con của nó theo khoảng đệm đã cho. Nó chứa **EdgeInsets** và **EdgeInsets.fromLTRB** cho phía mong muốn mà bạn muốn cung cấp đệm.

```
const Greetings(
  child: Padding(
    padding: EdgeInsets.all(14.0),
    child: Text('Hello Cafedev!'),
```

```
  ),
)
```

Center: widget này cho phép bạn căn giữa widget trong chính nó.

Align: Đây là một widget, căn chỉnh widget của nó trong chính nó và định kích thước nó dựa trên kích thước của đối tượng con. Nó cung cấp nhiều quyền kiểm soát hơn để đặt widget ở vị trí chính xác mà bạn cần.

```
Center(
  child: Container(
    height: 110.0,
    width: 110.0,
    color: Colors.blue,
    child: Align(
      alignment: Alignment.topLeft,
      child: FlutterLogo(
        size: 50,
      ),
    ),
  ),
)
```

SizedBox: widget này cho phép bạn cung cấp kích thước được chỉ định cho widget thông qua tất cả các màn hình.

```
SizedBox(
  width: 300.0,
  height: 450.0,
  child: const Card(child: Text('Hello Cafedev!')),
)
```

AspectRatio: widget này cho phép bạn giữ kích thước của widget theo một tỷ lệ khung hình được chỉ định.

```
AspectRatio(
  aspectRatio: 5/3,
  child: Container(
    color: Colors.bluel,


```

```
      ),
    ),
```

Baseline widget này thay đổi widget theo đường cơ sở của widget con bên trong.

```
child: Baseline(
          baseline: 30.0,
          baselineType: TextBaseline.alphabetic,
          child: Container(
               height: 60,
               width: 50,
               color: Colors.blue,
          ),
)
```

ConstrainedBox: Đây là một widget cho phép bạn buộc các ràng buộc bổ sung lên widget con của nó. Nó có nghĩa là bạn có thể buộc widget con có một ràng buộc cụ thể mà không làm thay đổi các thuộc tính của widget con.

```
ConstrainedBox(
  constraints: new BoxConstraints(
    minHeight: 150.0,
    minWidth: 150.0,
    maxHeight: 300.0,
    maxWidth: 300.0,
  ),
  child: new DecoratedBox(
    decoration: new BoxDecoration(color: Colors.red),
  ),
),
```

CustomSingleChildLayout: Nó là một widget, chuyển từ bố cục của con đơn thành một delegate. Người được ủy quyền(delegate) quyết định vị trí của widget và cũng được sử dụng để xác định kích thước của widget.

FittedBox: Nó chia tỷ lệ và định vị widget theo sự **phù hợp** được chỉ định .

```
import 'package:flutter/material.dart';

void main() => runApp(MyApp());

```

```
5
6   class MyApp extends StatelessWidget {
7     // It is the root widget of your application.
8     @override
9     Widget build(BuildContext context) {
10      return MaterialApp(
11        title: 'Multiple Layout Widget',
12        debugShowCheckedModeBanner: false,
13        theme: ThemeData(
14          // This is the theme of your application.
15          primarySwatch: Colors.green,
16        ),
17        home: MyHomePage(),
18      );
19    }
20  }
21  class MyHomePage extends StatelessWidget {
22
23    @override
24    Widget build(BuildContext context) {
25      return Scaffold(
26          appBar: AppBar(title: Text("FittedBox Widget")),
27          body: Center(
28          child: FittedBox(child: Row(
29            children: <Widget>[
30              Container(
31                child: Image.asset('assets/computer.png'),
32                ),
33                Container(
34                  child: Text("This is a widget"),
35                )
36              ],
37            ),
38            fit: BoxFit.contain,
39          )
40        ),
41      );
42    }
    }
```

Đầu ra

FractionallySizedBox: Nó là một widget cho phép kích thước của widget con của nó theo phần nhỏ của không gian có sẵn.

Chiều cao và chiều rộng **nội tại:** Chúng là một widget cho phép chúng ta định kích thước widget của nó theo chiều cao và chiều rộng nội tại của widget con.

LimitedBox: widget này cho phép chúng ta giới hạn kích thước của nó chỉ khi nó không bị giới hạn.

Offstage: Nó được sử dụng để đo kích thước của một widget mà không cần đưa nó lên màn hình.

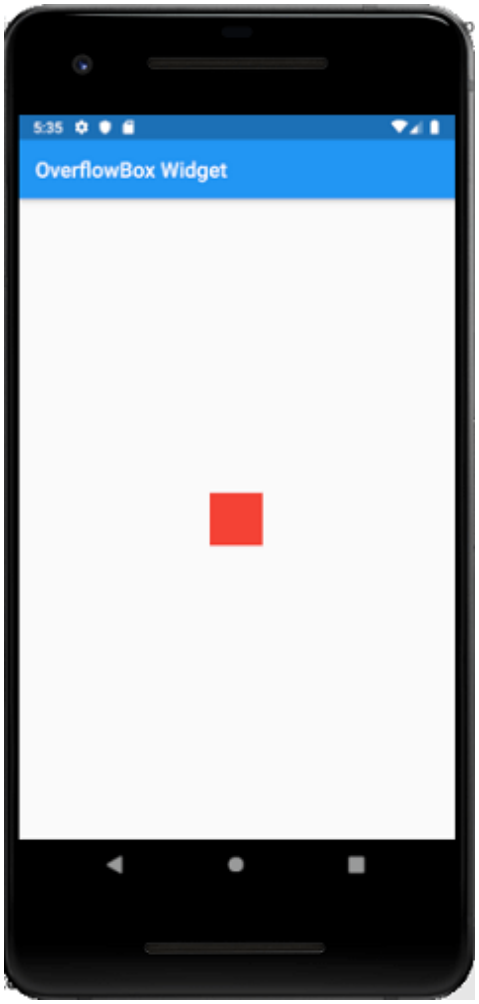
OverflowBox: Nó là một widget, cho phép áp đặt các ràng buộc khác nhau đối với widget con của nó so với nó nhận được từ cha mẹ. Nói cách khác, nó cho phép con làm tràn widget cha.

Thí dụ

```dart
import 'package:flutter/material.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  // It is the root widget of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Single Layout Widget',
      debugShowCheckedModeBanner: false,
      theme: ThemeData(
        // This is the theme of your application.
        primarySwatch: Colors.blue,
      ),
      home: MyHomePage(),
    );
  }
}
class MyHomePage extends StatelessWidget {

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("OverflowBox Widget"),
      ),
      body: Center(
      child: Container(
        height: 50.0,
        width: 50.0,
        color: Colors.red,
        child: OverflowBox(
          minHeight: 70.0,
          minWidth: 70.0,
          child: Container(
            height: 50.0,
            width: 50.0,
            color: Colors.blue,
            ),
          ),
        ),
      ),
    );

```

```
46        }
      }
```

Đầu ra



4. Đa widget

Đa widget là một loại widget chứa **nhiều hơn một widget con bên trong** và cách bố trí của các widget này là **duy nhất** . Ví dụ: widget Hàng bố trí widget theo hướng ngang và widget Cột bố trí widget theo hướng dọc. Nếu chúng ta kết hợp widget Hàng và Cột, thì nó có thể xây dựng bất kỳ cấp độ nào của widget phức tạp.

Ở đây, chúng ta sẽ tìm hiểu các loại khác nhau của nhiều widget:

Row: Nó cho phép sắp xếp các widget con của nó theo hướng nằm ngang.

Thí dụ

```dart
import 'package:flutter/material.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  // It is the root widget of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Multiple Layout Widget',
      debugShowCheckedModeBanner: false,
      theme: ThemeData(
        // This is the theme of your application.
        primarySwatch: Colors.blue,
      ),
      home: MyHomePage(),
    );
  }
}
class MyHomePage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Center(
      child: Container(
        alignment: Alignment.center,
        color: Colors.white,
        child: Row(
          children: <Widget>[
            Expanded(
              child: Text('Peter', textAlign: TextAlign.center),
            ),
            Expanded(
              child: Text('John', textAlign: TextAlign.center ),

            ),
            Expanded(
              child: FittedBox(
                fit: BoxFit.contain, // otherwise the logo will be t
                child: const FlutterLogo(),
              ),
            ),
          ],
        ),
      ),
```

```
46          );
47      }
    }
```

Đầu ra

Column: Nó cho phép sắp xếp các widget con của nó theo hướng dọc.

ListView: Đây là widget cuộn phổ biến nhất cho phép chúng ta sắp xếp các widget con của nó lần lượt theo hướng cuộn.

GridView: Nó cho phép chúng ta sắp xếp các widget con của nó dưới dạng một mảng widget 2D, có thể cuộn được. Nó bao gồm một mô hình lặp lại của các ô được sắp xếp theo bố cục ngang và dọc.

Expanded: Nó cho phép tạo các con của widget Hàng và Cột chiếm diện tích tối đa có thể.

Table: Nó là một widget cho phép chúng ta sắp xếp các con của nó trong một widget dựa trên bảng.

Flow: Nó cho phép chúng ta triển khai widget dựa trên luồng.

Stack: Đây là một widget thiết yếu, chủ yếu được sử dụng để chồng lên một số widget. Nó cho phép bạn đặt nhiều lớp lên màn hình. Ví dụ sau đây giúp bạn hiểu điều đó.

```
import 'package:flutter/material.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  // It is the root widget of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Multiple Layout Widget',
      debugShowCheckedModeBanner: false,
      theme: ThemeData(
        // This is the theme of your application.
        primarySwatch: Colors.blue,
      ),
      home: MyHomePage(),
    );
  }
}
class MyHomePage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Center(
      child: Container(
        alignment: Alignment.center,
        color: Colors.white,
        child: Stack(
          children: <Widget>[
            // Max Size
            Container(
              color: Colors.blue,
            ),
            Container(
              color: Colors.pink,
              height: 400.0,
```

```
              width: 300.0,
            ),
            Container(
              color: Colors.yellow,
              height: 220.0,
              width: 200.0,
            )
          ],
        ),
      ),
    );
  }
}
```

Đầu ra

5. Xây dựng bố cục phức tạp

Trong phần này, chúng ta sẽ tìm hiểu cách bạn có thể tạo giao diện người dùng phức tạp bằng cách sử dụng cả widget bố cục đơn và nhiều widget. Khung bố cục cho phép

bạn tạo bố cục giao diện người dùng phức tạp bằng cách lồng các hàng và cột vào bên trong các hàng và cột.

Hãy để cafedev tạo một ví dụ về giao diện người dùng phức tạp bằng cách tạo **danh sách sản phẩm** . Với mục đích này, trước tiên bạn cần thay thế code của tệp **main.dart** bằng đoạn mã sau.

```
import 'package:flutter/material.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  // It is the root widget of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo Application', theme: ThemeData(
      primarySwatch: Colors.green,),
      home: MyHomePage(title: 'Complex layout example'),
    );
  }
}
class MyHomePage extends StatelessWidget {
  MyHomePage({Key key, this.title}) : super(key: key);
  final String title;

  @override
  Widget build(BuildContext context) {
    return Scaffold(
        appBar: AppBar(title: Text("Product List")),
        body: ListView(
          padding: const EdgeInsets.fromLTRB(3.0, 12.0, 3.0, 12.0),
          children: <Widget>[
            ProductBox(
                name: "iPhone",
                description: "iPhone is the top branded phone ever",
                price: 55000,
                image: "iphone.png"
            ),
            ProductBox(
                name: "Android",
                description: "Android is a very stylish phone",
                price: 10000,
```

```
                    image: "android.png"
                ),
                ProductBox(
                    name: "Tablet",
                    description: "Tablet is a popular device for officia
                    price: 25000,
                    image: "tablet.png"
                ),
                ProductBox(
                    name: "Laptop",
                    description: "Laptop is most famous electronic devic
                    price: 35000,
                    image: "laptop.png"
                ),
                ProductBox(
                    name: "Desktop",
                    description: "Desktop is most popular for regular us
                    price: 10000,
                    image: "computer.png"
                ),
              ],
            )
        );
    }
}
class ProductBox extends StatelessWidget {
    ProductBox({Key key, this.name, this.description, this.price, this
          super(key: key);
    final String name;
    final String description;
    final int price;
    final String image;

    Widget build(BuildContext context) {
      return Container(
          padding: EdgeInsets.all(2),
          height: 110,
          child: Card(
              child: Row(
                  mainAxisAlignment: MainAxisAlignment.spaceEvenly,
                  children: <Widget>[
                    Image.asset("assets/" + image),
                    Expanded(
                        child: Container(
                            padding: EdgeInsets.all(5),
```

```
                            child: Column(
                              mainAxisAlignment: MainAxisAlignment.spa
                              children: <Widget>[
                                Text(
                                    this.name, style: TextStyle(
                                    fontWeight: FontWeight.bold
                                ),
                                ),
                                Text(this.description), Text(
                                    "Price: " + this.price.toString()
                                ),
                              ],
                            )
                        )
                    )
                ]
            )
        )
    );
  }
}
```

Trong đoạn code trên, chúng ta tạo **ProductBox** widget chứa các chi tiết của sản phẩm, chẳng hạn như hình ảnh, tên, giá và mô tả. Trong widget ProductBox, chúng tôi sử dụng các widget sau: Vùng chứa, Hàng, Cột, Mở rộng, Thẻ, Văn bản, Hình ảnh, v.v. widget này có bố cục sau:

Đầu ra

Bây giờ, khi chúng ta chạy file dart trong trình giả lập Android, nó sẽ cho kết quả như sau.

Cài ứng dụng cafedev để dễ dàng cập nhật tin và học lập trình mọi lúc mọi nơi tại đây.

Tài liệu từ cafedev:

- Full series tự học Flutter từ cơ bản tới nâng cao tại đây nha.
- **Các nguồn kiến thức MIỄN PHÍ VÔ GIÁ từ cafedev tại đây**

Nếu bạn thấy hay và hữu ích, bạn có thể tham gia các kênh sau của cafedev để nhận được nhiều hơn nữa:

- **Group Facebook**
- **Fanpage**
- **Youtube**
- **Instagram**
- **Twitter**
- **Linkedin**

- **Pinterest**
- **Trang chủ**

Chào thân ái và quyết thắng!

Đăng ký kênh youtube để ủng hộ Cafedev nha các bạn, Thanks you!

Bài trước

Tự học Flutter | Giới thiệu chi tiết về widgets và một số widgets cần phải biết trong Flutter

Bài tiếp theo

Tự học Flutter | Tìm hiểu về Cử chỉ(Gestures) với giao diện trong Flutter

David Xuân

https://cafedev.vn/

Nội dung chính

1. Widget Flutter

Trong phần này, chúng ta sẽ tìm hiểu khái niệm về một widget, cách tạo nó và các loại khác nhau của chúng có sẵn trong Flutter framework. Chúng ta đã biết trước đó rằng mọi thứ trong Flutter đều là một widget.

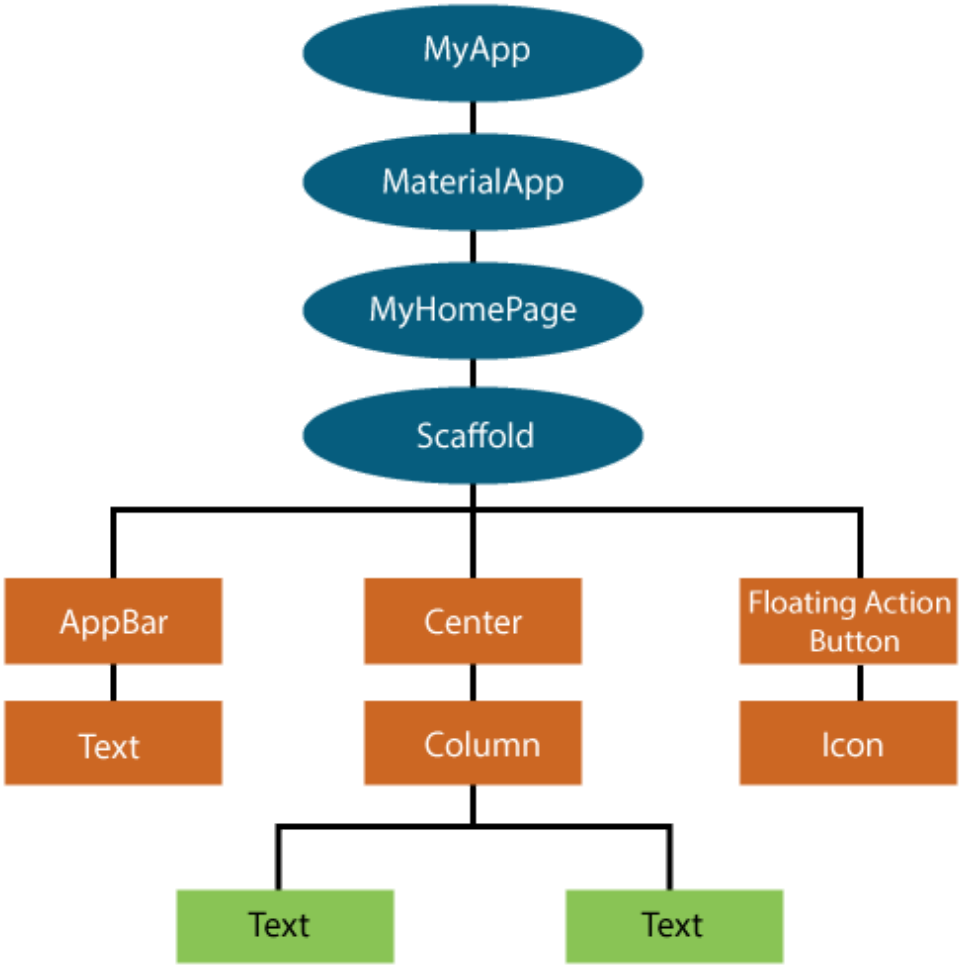
Nếu bạn đã quen thuộc với React hoặc Vue.js, thì bạn sẽ dễ dàng hiểu được Flutter.

Bất cứ khi nào bạn định viết mã để xây dựng bất cứ thứ gì trong Flutter, nó sẽ nằm trong một widget. Mục đích chính là xây dựng ứng dụng từ các widget. Nó mô tả chế độ xem ứng dụng của bạn trông như thế nào với cấu hình và trạng thái hiện tại của chúng. Khi bạn thực hiện bất kỳ thay đổi nào trong code, widget con sẽ xây dựng lại mô tả của

nó bằng cách tính toán sự khác biệt của widget con hiện tại và trước đó để xác định những thay đổi tối thiểu đối với việc hiển thị trong giao diện người dùng của ứng dụng.

Các widget được lồng vào nhau để xây dựng ứng dụng. Nó có nghĩa là thư mục gốc của ứng dụng của bạn tự nó là một widget, và tất cả các cách nhìn xuống cũng là một widget. Ví dụ: một widget có thể hiển thị một thứ gì đó, có thể xác định thiết kế, có thể xử lý tương tác, v.v.

Hình ảnh dưới đây là một mô tả trực quan đơn giản của cây widget.



Chúng ta có thể tạo widget Flutter như sau:

```
Class ImageWidget extends StatelessWidget {
          // Class Stuff
}
```

Ví dụ về Hello World

```dart
import 'package:flutter/material.dart';

class MyHomePage extends StatelessWidget {
  MyHomePage({Key key, this.title}) : super(key: key);
  // This widget is the home page of your application.
  final String title;

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(this.title),
      ),
      body: Center(
        child: Text('Hello World Cafedev'),
      ),
    );
  }
}
```

2. Các loại widget con

Chúng ta có thể chia widget Flutter thành hai loại:

1. Hiển thị(Visible) (Đầu ra và Đầu vào)
2. Vô hình(Invisible)(Bố cục và Kiểm soát – Layout and Control)

2.1 widget hiển thị

Các widget hiển thị có liên quan đến dữ liệu đầu vào và đầu ra của người dùng. Một số loại quan trọng của widget con này là:

Text

widget **Text** giữ một số văn bản để hiển thị trên màn hình. Chúng ta có thể căn chỉnh widget văn bản bằng cách sử dụng **thuộc** tính **textAlign** và thuộc tính style cho phép tùy chỉnh **Text** bao gồm phông chữ, độ đậm của phông chữ, kiểu phông chữ, khoảng cách giữa các chữ cái, màu sắc và nhiều hơn nữa. Chúng ta có thể sử dụng nó như các đoạn mã dưới đây.

```
Text(
'Hello, Cafedev!',
textAlign: TextAlign.center,
style: new TextStyle(fontWeight: FontWeight.bold),
)
```

Button

widget này cho phép bạn thực hiện một số hành động khi nhấp chuột. Flutter không cho phép bạn sử dụng trực tiếp widget TextButton; thay vào đó, nó sử dụng một loại **Button** như TextButton và OutlineButton . Chúng ta có thể sử dụng nó như các đoạn mã dưới đây.

```
 TextButton (
                    child: Text("TextButton With Background and Fore
                    onPressed: () {},
                    style: ButtonStyle(
                      backgroundColor: MaterialStateProperty.all<Col
                      foregroundColor: MaterialStateProperty.all<Col
                    )
                )

OutlineButton(
  child: new Text("Button text"),
  onPressed: null,
  shape: new RoundedRectangleBorder(borderRadius: new BorderRadius.c
)
```

Trong ví dụ trên, thuộc tính **onPressed** cho phép chúng ta thực hiện một hành động khi bạn nhấp vào **Button** và thuộc tính **elevation** được sử dụng để thay đổi mức độ nổi bật của nó.

Image

widget con này giữ hình ảnh có thể tìm nạp hình ảnh từ nhiều nguồn như từ thư mục nội dung hoặc trực tiếp từ URL. Nó cung cấp nhiều hàm tạo để tải hình ảnh, được đưa ra dưới đây:

- **Hình ảnh(Image):** Đây là một trình tải hình ảnh chung, được sử dụng bởi **ImageProvider** .
- **Tài sản(asset):** Nó tải hình ảnh từ thư mục tài sản dự án của bạn.
- **tệp(file):** Nó tải hình ảnh từ thư mục hệ thống.
- **bộ nhớ(memory):** Nó tải hình ảnh từ bộ nhớ.
- **mạng(network):** Nó tải hình ảnh từ mạng.

Để thêm hình ảnh vào dự án, trước tiên bạn cần tạo một thư mục nội dung nơi bạn lưu giữ hình ảnh của mình và sau đó thêm dòng bên dưới vào tệp **pubspec.yaml** .

```
assets:
  - assets/
```

Bây giờ, thêm dòng sau vào tệp dart.

```
mage.asset('assets/computer.png')
```

Mã nguồn hoàn chỉnh để thêm hình ảnh được hiển thị bên dưới trong ví dụ **hello world** .

```
class MyHomePage extends StatelessWidget {
  MyHomePage({Key key, this.title}) : super(key: key);
  // This widget is the home page of your application.
  final String title;

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(this.title),
      ),
      body: Center(
        child: Image.asset('assets/computer.png'),
      ),
    );
  }
}
```

Khi bạn chạy ứng dụng, nó sẽ đưa ra kết quả sau.

Icon

Widget này hoạt động như một thùng chứa để lưu trữ Biểu tượng trong Flutter. Đoạn mã sau đây giải thích rõ ràng hơn.

```
Icon(
  Icons.add,
  size: 34.0,
)
```

2.2 widget ẩn

Các widget vô hình có liên quan đến cách bố trí và kiểm soát các widget. Nó cung cấp việc kiểm soát cách các widget thực sự hoạt động và cách chúng sẽ hiển thị trên màn hình. Một số loại widget quan trọng là:

Column

Widget dạng cột là một loại widget sắp xếp tất cả các widget con của nó theo hàng dọc. Nó cung cấp khoảng cách giữa các Widget con bằng **mainAxisAlignment** và **crossAxisAlignment**. Trong các thuộc tính này, trục chính là trục tung, và trục chéo là trục hoành.

Thí dụ

Đoạn code dưới đây xây dựng hai phần tử widget con theo chiều dọc.

```
Column(
  mainAxisAlignment: MainAxisAlignment.center,
  children: <Widget>[
     Text(
      "VegElement",
    ),
     Text(
      "Non-vegElement"
    ),
  ],
),
```

Example

widget hàng tương tự như widget cột, nhưng nó xây dựng một widget con theo chiều ngang chứ không phải theo chiều dọc. Ở đây, trục chính là trục hoành, và trục chéo là trục tung.

Thí dụ

Các đoạn mã dưới đây xây dựng hai phần tử widget theo chiều ngang.

```
Row(
  mainAxisAlignment: MainAxisAlignment.spaceEvenly,
  children: <Widget>[
     Text(
      "VegElement",
    ),
     Text(
```

```
        "Non-vegElement"
      ),
    ],
  ),
```

Center

widget con này được sử dụng để căn giữa widget con, nằm bên trong nó. Tất cả các ví dụ trước đều chứa bên trong widget **Center**.

Thí dụ

```
Center(
  child: new clumn(
    mainAxisAlignment: MainAxisAlignment.spaceEvenly,
    children: <Widget>[
       Text(
        "VegElement",
      ),
       Text(
        "Non-vegElement"
      ),
    ],
  ),
),
```

Padding

widget này bao bọc các widget khác để cung cấp cho chúng phần đệm theo các hướng được chỉ định. Bạn cũng có thể cung cấp đệm theo mọi hướng. Chúng ta có thể hiểu nó từ ví dụ dưới đây cung cấp cho phần đệm widget văn bản là 6.0 theo mọi hướng.

Thí dụ

```
Padding(
  padding: const EdgeInsets.all(6.0),
  child: Text(
    "Element 1",
  ),
),
```

Scaffold

widget này cung cấp một framework cho phép bạn thêm các yếu tố thiết kế material design phổ biến như AppBar, Floating Action Buttons, Drawers, v.v.

Stack

Nó là một widget thiết yếu, chủ yếu được sử dụng để **chồng lên** một widget, chẳng hạn như một nút trên nền gradient.

3. Quản lý Widget State

Trong Flutter, chủ yếu có hai loại widget:

- StatelessWidget
- StatefulWidget

3.1 StatefulWidget

StatefulWidget có thông tin state. Nó chủ yếu chứa hai lớp: **state object** và **widget**. Nó là động vì nó có thể thay đổi dữ liệu bên trong trong suốt thời gian tồn tại của widget. widget con này không có phương thức **build()** . Nó có phương thức **createState()** , trả về một lớp mở rộng Lớp **state của** Flutters. Các ví dụ của StatefulWidget là Checkbox, Radio, Slider, InkWell, Form và TextField.

Thí dụ

```
class Car extends StatefulWidget {
  const Car({ Key key, this.title }) : super(key: key);

  @override
  _CarState createState() => _CarState();
}

class _CarState extends State<Car> {
  @override
  Widget build(BuildContext context) {
    return Container(
      color: const Color(0xFEEFE),

```

```
            child: Container(
              child: Container( //child: Container( ) )
            )
        );
    }
}
```

3.2 StatelessWidget

StatelessWidget không có bất kỳ thông tin trạng thái nào. Nó vẫn tĩnh trong suốt vòng đời của nó. Các ví dụ về StatelessWidget là Văn bản, Hàng, Cột, Vùng chứa, v.v.

Thí dụ

```
class MyStatelessCarWidget extends StatelessWidget {
  const MyStatelessCarWidget ({ Key key }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Container(color: const Color(0x0xFEEFE));
  }
}
```

Cài ứng dụng cafedev để dễ dàng cập nhật tin và học lập trình mọi lúc mọi nơi tại đây.

Tài liệu từ cafedev:

- Full series tự học Flutter từ cơ bản tới nâng cao tại đây nha.
- **Các nguồn kiến thức MIỄN PHÍ VÔ GIÁ từ cafedev tại đây**

Nếu bạn thấy hay và hữu ích, bạn có thể tham gia các kênh sau của cafedev để nhận được nhiều hơn nữa:

- **Group Facebook**
- **Fanpage**
- **Youtube**
- **Instagram**

- **Twitter**
- **Linkedin**
- **Pinterest**
- **Trang chủ**

Chào thân ái và quyết thắng!

Đăng ký kênh youtube để ủng hộ Cafedev nha các bạn, Thanks you!

Bài trước

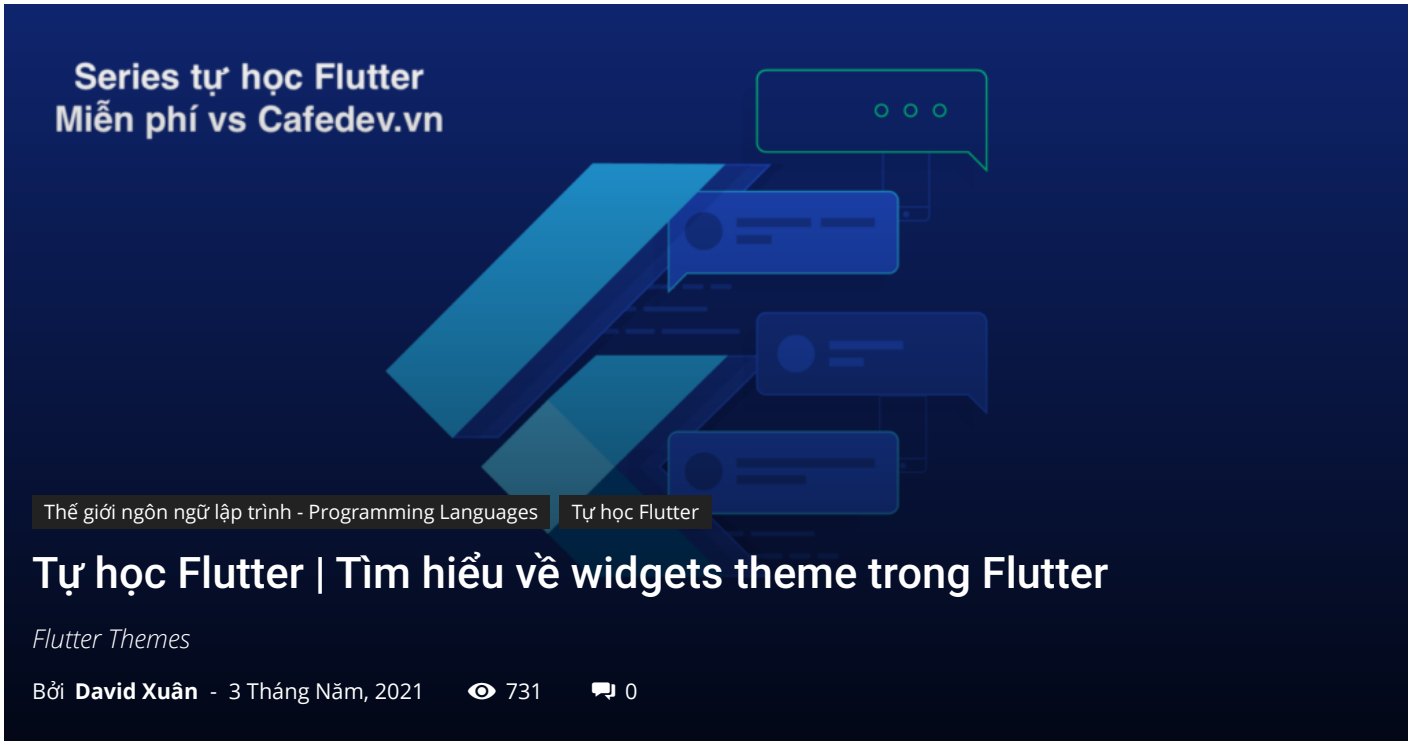
Tự học Flutter | Giới thiệu lập trình với ngôn ngữ Dart

Bài tiếp theo

Tự học Flutter | Tìm hiểu về bố cụ(layout) giao diện trong Flutter

David Xuân

https://cafedev.vn/



Tự học Flutter | Tìm hiểu về widgets theme trong Flutter

Flutter Themes

Bởi **David Xuân** - 3 Tháng Năm, 2021 731 0

Chủ đề(theme) là các gói đặt trước chứa các **giao diện đồ họa trên** trang web hoặc màn hình ứng dụng dành cho thiết bị di động của chúng ta. Nó làm cho giao diện người dùng hấp dẫn hơn. **Chúng ta chủ yếu sử dụng các chủ đề để chia sẻ màu sắc và kiểu phông chữ** trong toàn bộ ứng dụng.

Trong quá trình phát triển thiết bị di động, việc thêm chủ đề **Ánh sáng(Light)** và **Bóng tối(Dark)** cho ứng dụng của chúng ta trở nên bắt buộc . Ngày nay, hầu hết mọi người thích phiên bản tối của chủ đề hơn chủ đề phiên bản sáng vì nó giúp họ dễ chịu hơn và tăng tuổi thọ pin.

Trong Flutter , chúng ta có thể sử dụng **Theme widgets** chứa màu sắc và kiểu phông chữ cho một khu vực cụ thể của ứng dụng hoặc xác định **các** chủ đề **trên toàn ứng dụng** . Các chủ đề trên toàn ứng dụng cũng là các tiện ích Chủ đề, được tạo trong thư mục gốc của ứng dụng của chúng ta trong tiện ích **MaterialApp** .

Sau khi xác định chủ đề, chúng ta có thể sử dụng chủ đề đó trong bất kỳ tiện ích nào ở bất kỳ nơi nào chúng ta cần trong ứng dụng. Các tiện ích vật liệu trong Flutter cũng có

thể sử dụng Chủ đề của chúng ta để đặt kiểu phông chữ và màu nền cho Thanh ứng dụng, Nút, Nút, Hộp kiểm và nhiều thứ khác.

Flutter sử dụng chủ đề mặc định trong khi tạo ứng dụng. Nếu muốn chia sẻ chủ đề tùy chỉnh cho toàn bộ ứng dụng, chúng ta cần sử dụng **Dữ liệu** chủ đề trong tiện ích MateialApp ().

Đôi khi chúng ta muốn ghi đè chủ đề của toàn ứng dụng trong một phần của ứng dụng. Trong trường hợp đó, chúng ta cần bao bọc phần của ứng dụng trong tiện ích chủ đề. Flutter cho phép chúng ta hai cách tiếp cận để thực hiện điều này:

1. Bằng cách tạo một ThemeData
2. Bằng cách mở rộng chủ đề mẹ

1. Bằng cách tạo một ThemeData

Cách tiếp cận đầu tiên được sử dụng khi chúng ta không muốn kế thừa bất kỳ màu ứng dụng hoặc kiểu phông chữ nào. Trong trường hợp đó, chúng ta sẽ tạo một phiên bản của ThemeData() và chuyển nó vào widget Chủ đề, như được hiển thị trong đoạn mã dưới đây:

```
Theme(
  data: ThemeData(
    accentColor: Colors.blue,
  ),
  child: FloatingActionButton(
    onPressed: () {},
    child: Icon(Icons.person),
  ),
);
```

2. Bằng cách mở rộng chủ đề mẹ

Nếu bạn không muốn ghi đè bất kỳ thứ gì, hãy sử dụng cách tiếp cận thứ hai mở rộng chủ đề chính. Nó có thể được xử lý bằng cách sử dụng phương thức **copyWith ()** . Xem đoạn mã dưới đây:

```
Theme(
  data: Theme.of(context).copyWith(accentColor: Colors.blue),
  child: FloatingActionButton(
    onPressed: null,
    child: Icon(Icons.person),
  ),
);
```

3. Cách sử dụng Chủ đề

Sau khi xác định một chủ đề, chúng ta có thể sử dụng nó vào các phương thức **build () widget** với phương thức **Theme.of (context)** . Phương thức này nhìn vào cây widget và trả về chủ đề đầu tiên trong cây. Nếu bạn chưa xác định widget của mình, chủ đề của ứng dụng sẽ được trả lại.

Trong đoạn mã dưới đây, **FloatingActionButton** sử dụng kỹ thuật này để trả về **màu sắc** .

```
Container(
  color: Theme.of(context).accentColor,
  child: Text(
    'Text with a background color',
    style: Theme.of(context).textTheme.headline,
  ),
);
```

Hãy để chúng ta hiểu cách sử dụng ThemeData trong ứng dụng Flutter thông qua ví dụ dưới đây.

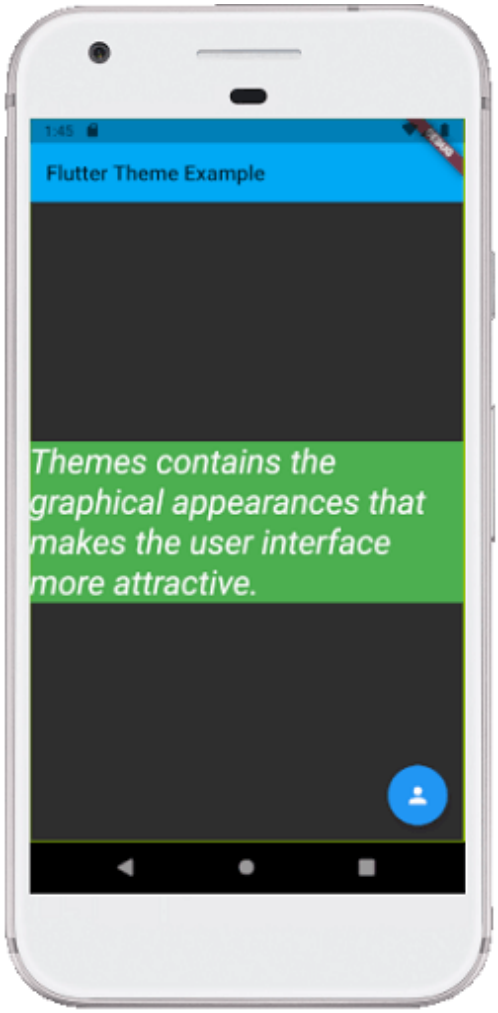
```
import 'package:flutter/material.dart';

void main() {runApp(MyApp());}

class MyApp extends StatelessWidget {
  @override
```

```
  Widget build(BuildContext context) {
    return MaterialApp(
      theme: ThemeData(
        // Define the default brightness and colors.
        brightness: Brightness.dark,
        primaryColor: Colors.lightBlue,
        accentColor: Colors.green,

        // Define the default font family.
        fontFamily: 'Monotype Coursiva',

        // Define the TextTheme that specifies the default
        // text styling for headlines, titles, bodies of text, and m
        textTheme: TextTheme(
          headline: TextStyle(fontSize: 32.0, fontStyle: FontStyle.i
        ),
      ),
      home: MyThemePage(),
    );
  }
}

class MyThemePage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Flutter Theme Example'),
      ),
      body: Center(
        child: Container(
          color: Theme.of(context).accentColor,
          child: Text(
            'Themes contains the graphical appearances that makes th
            style: Theme.of(context).textTheme.headline,
          ),
        ),
      ),
      floatingActionButton: Theme(
        data: Theme.of(context).copyWith(
          colorScheme:
          Theme.of(context).colorScheme.copyWith(secondary: Colors.b
        ),
        child: FloatingActionButton(
          onPressed: null,
```

```
            child: Icon(Icons.person),
          ),
        ),
      );
    }
  }
```

Đầu ra:

Khi chúng ta chạy ứng dụng trong thiết bị hoặc trình giả lập, chúng ta sẽ thấy giao diện người dùng tương tự như ảnh chụp màn hình bên dưới.



Cài ứng dụng cafedev để dễ dàng cập nhật tin và học lập trình mọi lúc mọi nơi tại đây.

Tài liệu từ cafedev:

- Full series tự học Flutter từ cơ bản tới nâng cao tại đây nha.

- **Các nguồn kiến thức MIỄN PHÍ VÔ GIÁ từ cafedev tại đây**

Nếu bạn thấy hay và hữu ích, bạn có thể tham gia các kênh sau của cafedev để nhận được nhiều hơn nữa:

- **Group Facebook**
- **Fanpage**
- **Youtube**
- **Instagram**
- **Twitter**
- **Linkedin**
- **Pinterest**
- **Trang chủ**

Chào thân ái và quyết thắng!

Đăng ký kênh youtube để ủng hộ Cafedev nha các bạn, Thanks you!

Bài trước

Tự học Flutter | Tìm hiểu về widget Bottom Navigation Bar trong Flutter

Bài tiếp theo

Tự học Flutter | Tìm hiểu về widget Table trong Flutter

David Xuân

https://cafedev.vn/

Thế giới ngôn ngữ lập trình - Programming Languages | Tự học Flutter

Tự học Flutter | Tìm hiểu về cách quản lý state(trạng thái) trong Flutter

Flutter State Management

Bởi **David Xuân** - 19 Tháng Tư, 2021 1273 0

Trong phần này, chúng ta sẽ thảo luận về quản lý state(trạng thái) và cách chúng ta có thể xử lý nó trong Flutter. Chúng ta biết rằng trong Flutter, mọi thứ đều là một widget. Có thể phân loại widget này thành hai loại, một là **widget Không trạng(Stateless widget) thái** và một là **widget có Trạng thái(Stateful widget).** widget không trạng thái không có bất kỳ trạng thái bên trong nào. Nó có nghĩa là một khi nó được xây dựng, chúng ta không thể thay đổi hoặc sửa đổi nó cho đến khi chúng được khởi tạo lại. Mặt khác, widget Stateful là động và có trạng thái. Nó có nghĩa là chúng ta có thể sửa đổi nó một cách dễ dàng trong suốt vòng đời của nó mà không cần khởi động lại nó một lần nữa.

Nội dung chính

1. State(Trạng thái) là gì?

Trạng thái là thông tin có thể **đọc** được khi widget được tạo và có thể **thay đổi hoặc sửa đổi** trong suốt thời gian tồn tại của ứng dụng. Nếu bạn muốn thay đổi widget của mình, bạn cần cập nhật đối tượng trạng thái, có thể được thực hiện bằng cách sử dụng hàm setState() có sẵn cho các widget Stateful. Hàm **setState()** cho phép chúng ta thiết lập các thuộc tính của **đối tượng** trạng thái kích hoạt vẽ lại giao diện người dùng.

Quản lý state(trạng thái) là một trong những quy trình phổ biến và cần thiết nhất trong vòng đời của một ứng dụng. Theo tài liệu chính thức, Flutter mang tính chất khai báo. Điều đó có nghĩa là Flutter xây dựng giao diện người dùng của mình bằng cách phản ánh trạng thái hiện tại của ứng dụng của bạn. Hình sau giải thích rõ hơn về nơi bạn có thể xây dựng giao diện người dùng từ trạng thái ứng dụng.

Chúng ta hãy lấy một ví dụ đơn giản để hiểu khái niệm quản lý state(trạng thái). Giả sử bạn đã tạo danh sách khách hàng hoặc sản phẩm trong ứng dụng của mình. Bây giờ, giả sử bạn đã thêm động một khách hàng hoặc sản phẩm mới vào danh sách đó. Sau đó, cần phải làm mới danh sách để xem mục mới được thêm vào bản ghi. Vì vậy, bất cứ khi nào bạn thêm một mục mới, bạn cần phải làm mới danh sách. Kiểu lập trình này yêu cầu quản lý state(trạng thái) xử lý tình huống như vậy để cải thiện hiệu suất. Đó là bởi vì mỗi khi bạn thực hiện một thay đổi hoặc cập nhật giống nhau, trạng thái sẽ được làm mới.

Trong Flutter , quản lý state(trạng thái) phân thành hai loại khái niệm, được đưa ra dưới đây:

1. Trạng thái tức thời(Ephemeral State)
2. Trạng thái ứng dụng(App State)

1.1 Trạng thái tức thời

Trạng thái này còn được gọi là Trạng thái giao diện người dùng hoặc trạng thái địa phương. Nó là một loại trạng thái có liên quan đến **widget cụ thể,** hoặc bạn có thể nói rằng nó là một trạng thái chứa trong một widget duy nhất. Trong loại trạng thái này,

bạn không cần sử dụng các kỹ thuật quản lý state(trạng thái). Ví dụ phổ biến của trạng thái này là **Text Field** .

Thí dụ

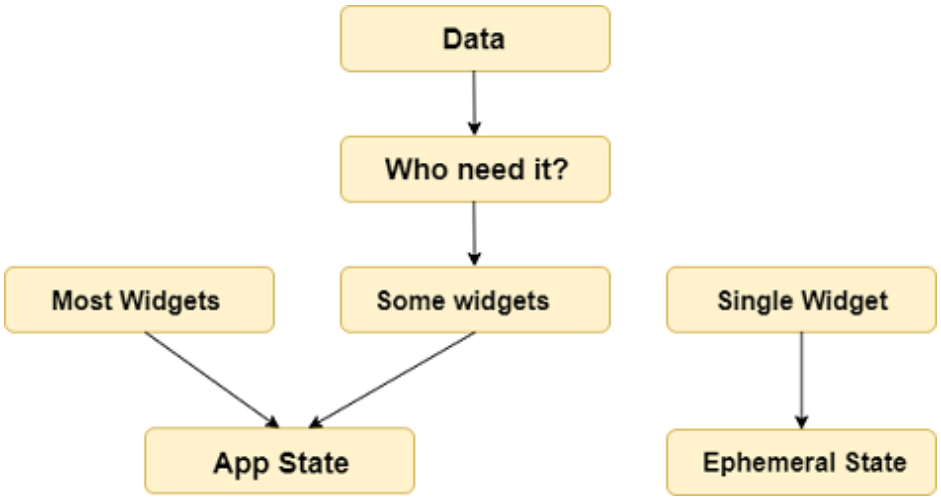
```
class MyHomepage extends StatefulWidget {
  @override
  MyHomepageState createState() => MyHomepageState();
}

class MyHomepageState extends State<MyHomepage> {
  String _name = "Peter";

  @override
  Widget build(BuildContext context) {
    return RaisedButton(
        child: Text(_name),
        onPressed: () {
           setState(() {
              _name = _name == "Peter" ? "John" : "Peter";
           });
         },
      );
  }
}
```

Trong ví dụ trên, **_name** là một trạng thái tạm thời. Ở đây, chỉ có hàm setState () bên trong lớp của StatefulWidget mới có thể truy cập vào _name. Phương thức xây dựng gọi một hàm setState (), hàm này thực hiện sửa đổi các biến trạng thái. Khi phương thức này được thực thi, đối tượng widget sẽ được thay thế bằng đối tượng mới, mang lại giá trị biến được sửa đổi.

1.2 Trạng thái ứng dụng

Nó khác với trạng thái tức thời. Đó là một loại trạng thái mà chúng ta muốn **chia sẻ** trên các phần khác nhau của ứng dụng và muốn giữ lại giữa các phiên của người dùng. Do đó, loại trạng thái này có thể được sử dụng trên toàn cầu. Đôi khi nó còn được gọi là trạng thái ứng dụng hoặc trạng thái chia sẻ. Một số ví dụ về trạng thái này là Tùy chọn người dùng, Thông tin đăng nhập, thông báo trong ứng dụng mạng xã hội, giỏ hàng trong ứng dụng thương mại điện tử, trạng thái đã đọc / chưa đọc của các bài báo trong ứng dụng tin tức, v.v.

Sơ đồ sau giải thích sự khác biệt giữa trạng thái tạm thời và trạng thái ứng dụng một cách phù hợp hơn.



Ví dụ đơn giản nhất về quản lý trạng thái ứng dụng có thể được học bằng cách sử dụng **gói nhà cung cấp.** Việc quản lý state(trạng thái) với nhà cung cấp rất dễ hiểu và ít phải viết mã. Nhà cung cấp là thư viện **của bên thứ ba** . Ở đây, chúng ta cần hiểu ba khái niệm chính để sử dụng thư viện này.

1. ChangeNotifier
2. ChangeNotifierProvider
3. Khách hàng(Consumer)

1.3 ChangeNotifier

ChangeNotifier là một lớp đơn giản, cung cấp thông báo thay đổi cho người nghe của nó. Nó dễ hiểu, dễ thực hiện và được tối ưu hóa cho một số lượng nhỏ người nghe. Nó được sử dụng để người nghe quan sát một mô hình để thay đổi. Trong điều này, chúng ta chỉ sử dụng phương **thứctifyListener()** để thông báo cho người nghe.

Ví dụ: chúng ta hãy xác định một mô hình dựa trên ChangeNotifier. Trong mô hình này, **Bộ đếm** được mở rộng với ChangeNotifier, được sử dụng để thông báo cho người nghe của nó khi chúng ta gọi InformListists(). Đây là phương thức duy nhất cần triển khai trong mô hình ChangeNotifier. Trong ví dụ này, chúng ta đã khai báo hai hàm là **tăng** và **giảm,** được sử dụng để tăng và giảm giá trị. Chúng ta có thể gọi phương thức notifyListeners() bất kỳ lúc nào mô hình thay đổi theo cách có thể thay đổi giao diện người dùng của ứng dụng của bạn.

```dart
import 'package:flutter/material.dart';

class Counter with ChangeNotifier {
  int _counter;

  Counter(this._counter);

  getCounter() => _counter;
  setCounter(int counter) => _counter = counter;

  void increment() {
    _counter++;
    notifyListeners();
  }

  void decrement() {
    _counter--;
    notifyListeners();
  }
}
```

ChangeNotifierProvider

ChangeNotifierProvider là widget cung cấp một **phiên bản** của ChangeNotifier cho con của nó. Nó đến từ gói nhà cung cấp. Các đoạn code sau đây giúp hiểu khái niệm về ChangeNotifierProvider.

Ở đây, chúng ta đã định nghĩa một **người xây dựng(builder)** những người sẽ tạo một đối tượng mới của **Counter** mô hình. ChangeNotifierProvider không xây dựng lại Bộ đếm trừ khi có nhu cầu cho việc này. Nó cũng sẽ tự động gọi phương thức **dispose()** trên mô hình Counter khi thể hiện không còn cần thiết nữa.

```dart
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      theme: ThemeData(
        primarySwatch: Colors.indigo,
      ),
      home: ChangeNotifierProvider<CounterModel>(
        builder: (_) => CounterModel(),
        child: CounterView(),
```

```
        ),
      );
    }
  }
```

Nếu có nhu cầu cung cấp nhiều hơn một lớp, bạn có thể sử dụng **MultiProvider.** MultiProvider là danh sách tất cả các Nhà cung cấp khác nhau đang được sử dụng trong phạm vi của nó. Nếu không sử dụng điều này, chúng tôi sẽ phải lồng các Nhà cung cấp của mình với một nhà cung cấp là con của người khác và người khác. Chúng ta có thể hiểu điều này từ đoạn mã dưới đây.

```
void main() {
  runApp(
    MultiProvider(
      providers: [
        ChangeNotifierProvider(builder: (context) => Counter()),
        Provider(builder: (context) => SomeOtherClass()),
      ],
      child: MyApp(),
    ),
  );
}
```

1.4 Consumer

Nó là một loại nhà cung cấp không làm bất kỳ công việc cầu kỳ. Nó chỉ gọi nhà cung cấp trong một widget con mới và ủy quyền triển khai bản dựng của nó cho người xây dựng. Đoạn code sau giải thích rõ ràng hơn. /P>

```
return Consumer<Counter>(
  builder: (context, count, child) {
    return Text("Total price: ${count.total}");
  },
);
```

Trong ví dụ trên, bạn có thể thấy rằng **widget consumer** chỉ yêu cầu một hàm trình tạo, được gọi bất cứ khi nào ChangeNotifier thay đổi. Hàm trình tạo chứa **ba** đối số, đó là **ngữ cảnh, số đếm và con(context, count, and child).** Đối số đầu tiên, ngữ cảnh chứa trong mọi phương thức build(). Đối số thứ hai là thể hiện của ChangeNotifier và

đối số thứ ba là đối số con được sử dụng để tối ưu hóa. Ý tưởng tốt nhất là đặt widget **consumer** càng sâu trong cây càng tốt.

Cài ứng dụng cafedev để dễ dàng cập nhật tin và học lập trình mọi lúc mọi nơi tại đây.

Tài liệu từ cafedev:

- Full series tự học Flutter từ cơ bản tới nâng cao tại đây nha.
- **Các nguồn kiến thức MIỄN PHÍ VÔ GIÁ từ cafedev tại đây**

Nếu bạn thấy hay và hữu ích, bạn có thể tham gia các kênh sau của cafedev để nhận được nhiều hơn nữa:

- **Group Facebook**
- **Fanpage**
- **Youtube**
- **Instagram**
- **Twitter**
- **Linkedin**
- **Pinterest**
- **Trang chủ**

Chào thân ái và quyết thắng!

Đăng ký kênh youtube để ủng hộ Cafedev nha các bạn, Thanks you!

Bài trước

Tự học Flutter | Tìm hiểu về Cử chỉ(Gestures) với giao diện trong Flutter

Bài tiếp theo

Tự học Flutter | Tìm hiểu về các IDE dùng để code Flutter

David Xuân

https://cafedev.vn/

Thế giới ngôn ngữ lập trình - Programming Languages | Tự học Flutter

Tự học Flutter | Tìm hiểu về Cử chỉ(Gestures) với giao diện trong Flutter

Flutter Gestures

Bởi **David Xuân** - 19 Tháng Tư, 2021 1044 0

Cử chỉ(Gestures) là một tính năng thú vị trong Flutter cho phép chúng ta tương tác với ứng dụng di động (hoặc bất kỳ thiết bị dựa trên cảm ứng). Nói chung, cử chỉ xác định bất kỳ hành động hoặc chuyển động vật lý nào của người dùng nhằm mục đích kiểm soát thiết bị di động. Một số ví dụ về cử chỉ là:

- Khi màn hình di động bị khóa, bạn trượt ngón tay trên màn hình để mở khóa.
- Nhấn vào một nút trên màn hình điện thoại di động của bạn và
- Nhấn và giữ biểu tượng ứng dụng trên thiết bị dựa trên cảm ứng để kéo biểu tượng đó qua các màn hình.

Chúng ta sử dụng tất cả những cử chỉ này trong cuộc sống hàng ngày để tương tác với điện thoại hoặc thiết bị dựa trên cảm ứng của bạn.

Flutter chia hệ thống cử chỉ thành hai lớp khác nhau, được đưa ra dưới đây:

1. Con trỏ(Pointers)
2. Cử chỉ(Gestures)

Nội dung chính

1. Con trỏ

Con trỏ(Pointers) là lớp đầu tiên đại diện cho dữ liệu thô về tương tác của người dùng. Nó có các sự kiện, mô tả **vị trí** và **chuyển động** của các con trỏ như chạm, chuột và kiểu trên màn hình. Flutter không cung cấp bất kỳ cơ chế nào để hủy hoặc dừng các sự kiện con trỏ được gửi đi thêm. Flutter cung cấp một widget **Listener** để lắng nghe các sự kiện con trỏ trực tiếp từ lớp widget. Con trỏ-sự kiện được phân loại thành bốn loại chủ yếu:

- PointerDownEvents
- PointerMoveEvents
- PointerUpEvents
- PointerCancelEvents

PointerDownEvents: Nó cho phép con trỏ tiếp xúc với màn hình tại một vị trí cụ thể.

PointerMoveEvents: Nó cho phép con trỏ di chuyển từ vị trí này đến vị trí khác trên màn hình.

PointerUpEvents: Nó cho phép con trỏ dừng tiếp xúc với màn hình.

PointerCancelEvents: Sự kiện này được gửi khi tương tác với con trỏ bị hủy.

2. Cử chỉ

Đây là lớp thứ hai đại diện cho **các hành động ngữ nghĩa** như chạm, kéo và chia tỷ lệ, được nhận dạng từ nhiều sự kiện con trỏ riêng lẻ. Nó cũng có thể gửi nhiều sự kiện tương ứng với vòng đời cử chỉ như kéo bắt đầu, kéo cập nhật và kéo kết thúc. Một số cử chỉ được sử dụng phổ biến được liệt kê dưới đây:

Chạm(Tap): Có nghĩa là chạm vào bề mặt màn hình từ đầu ngón tay trong một thời gian ngắn rồi thả chúng ra. Cử chỉ này chứa các sự kiện sau:

- onTapDown
- onTapUp
- onTap
- onTapCancel

Nhấn đúp(Double Tap): Nó tương tự như cử chỉ Nhấn, nhưng bạn cần nhấn hai lần trong thời gian ngắn. Cử chỉ này chứa các sự kiện sau:

- onDoubleTap

Kéo(Drag): Nó cho phép chúng ta chạm vào bề mặt của màn hình bằng đầu ngón tay và di chuyển nó từ vị trí này sang vị trí khác rồi thả chúng ra. Flutter phân loại kéo thành hai loại:

1. **Kéo ngang:** Cử chỉ này cho phép con trỏ di chuyển theo hướng ngang. Nó chứa các sự kiện sau:
   - onHorizontalDragStart
   - onHorizontalDragUpdate
   - onHorizontalDragEnd
2. **Kéo dọc:** Cử chỉ này cho phép con trỏ di chuyển theo hướng thẳng đứng. Nó chứa các sự kiện sau:
   - onVerticalDragStart
   - onVerticalDragStart
   - onVerticalDragStart

Nhấn lâu(Long Press): Có nghĩa là chạm vào bề mặt của màn hình tại một vị trí cụ thể trong một thời gian dài. Cử chỉ này chứa các sự kiện sau:

- onLongPress

Di chuyển(**Pan**) **:** Có nghĩa là chạm vào bề mặt của màn hình bằng đầu ngón tay, có thể di chuyển theo bất kỳ hướng nào mà không cần nhả đầu ngón tay. Cử chỉ này chứa các sự kiện sau:

- onPanStart

- onPanUpdate
- onPanEnd

Chụm(Pinch): Có nghĩa là chụm (di chuyển ngón tay và ngón cái của một người hoặc đưa chúng lại gần nhau trên màn hình cảm ứng) bề mặt của màn hình bằng cách sử dụng hai ngón tay để phóng to hoặc thu nhỏ màn hình.

3. Dò cử chỉ

Flutter cung cấp một tiện ích hỗ trợ tuyệt vời cho tất cả các loại cử chỉ bằng cách sử dụng tiện ích **GestureDetector** . GestureWidget là các widget không trực quan, chủ yếu được sử dụng để phát hiện cử chỉ của người dùng. Ý tưởng cơ bản của bộ phát hiện cử chỉ là một tiện ích **không trạng thái** có chứa các tham số trong hàm tạo của nó cho các sự kiện chạm khác nhau.

Trong một số tình huống, có thể có nhiều bộ phát hiện cử chỉ tại một vị trí cụ thể trên màn hình và sau đó, khung sẽ xác định cử chỉ nào sẽ được gọi. Widget GestureDetector quyết định cử chỉ nào sẽ nhận ra dựa trên lệnh gọi lại nào của nó là không rỗng.

Hãy để chúng ta tìm hiểu cách chúng ta có thể sử dụng các cử chỉ này trong ứng dụng của mình với sự kiện **onTap()** đơn giản và xác định cách GestureDetector xử lý điều này. Ở đây, chúng ta sẽ tạo một **tiện ích hộp,** thiết kế nó theo đặc điểm kỹ thuật mong muốn của chúng ta và sau đó thêm hàm onTap() vào nó.

Bây giờ, hãy tạo một dự án Flutter mới và thay thế mã sau trong tệp **main.dart** .

```
import 'package:flutter/material.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo Application', theme: ThemeData(
      primarySwatch: Colors.green,),
      home: MyHomePage(),
    );
  }
```

```
}
class MyHomePage extends StatefulWidget {
  @override
  MyHomePageState createState() => new MyHomePageState();
}

class MyHomePageState extends State<MyHomePage> {
  @override
  Widget build(BuildContext context) {
    return new Scaffold(
      appBar: new AppBar(
        title: new Text('Gestures Example'),
        centerTitle: true,
      ),
      body: new Center(child: GestureDetector(
          onTap: () {
            print('Box Clicked');
          },
          child: Container(
            height: 60.0,
            width: 120.0,
            padding: EdgeInsets.all(10.0),
            decoration: BoxDecoration(
              color: Colors.blueGrey,
              borderRadius: BorderRadius.circular(15.0),
            ),
            child: Center(child: Text('Click Me')),
          )
      )),
    );
  }
}
```

Đầu ra

Khi bạn chạy tệp dart này trong Android Studio, nó sẽ đưa ra kết quả sau trong trình giả lập.

Trong hình trên, bạn có thể thấy một nút có các cạnh tròn ở giữa màn hình. Khi bạn nhấn vào nút này, nó hoạt động giống như một nút và có thể thấy đầu ra trong bảng điều khiển.

Flutter cũng cung cấp một tập hợp các widget có thể cho phép bạn thực hiện một cử chỉ cụ thể cũng như nâng cao. Các tiện ích này được đưa ra dưới đây:

Loại bỏ(Dismissible): Đây là một loại widget hỗ trợ cử chỉ vuốt nhẹ để loại bỏ widget.

Draggable: Là một loại widget hỗ trợ cử chỉ kéo để di chuyển widget.

LongPressDraggable: Là một loại widget hỗ trợ cử chỉ kéo để di chuyển widget cùng với widget cha của nó.

DragTarget: Đây là một loại widget có thể chấp nhận bất kỳ widget **Draggable** nào

DubrePointer: Là một loại widget ẩn widget và các con của nó khỏi quá trình phát hiện cử chỉ.

AbsorbPointer: Đây là một loại widget tự dừng quá trình phát hiện cử chỉ. Do đó, bất kỳ widget chồng chéo nào đều không thể tham gia vào quá trình phát hiện cử chỉ và do đó, không có sự kiện nào được đưa ra.

Scrollable: Đây là một loại widget hỗ trợ cuộn nội dung có sẵn bên trong widget.

4. Ví dụ nhiều cử chỉ

Trong phần này, chúng ta sẽ xem nhiều cử chỉ hoạt động như thế nào trong các ứng dụng rung. Ứng dụng demo này bao gồm hai vùng chứa chính và con. Ở đây, mọi thứ được xử lý thủ công bằng cách sử dụng 'RawGestureDetector' và 'GestureRecognizer' tùy chỉnh GestureRecognizer tùy chỉnh cung cấp thuộc tính 'AllowMultipleGestureRecognizer' vào danh sách cử chỉ và tạo một 'GestureRecognizerFactoryWithHandlers'. Tiếp theo, khi sự kiện **onTap ()** được gọi, nó sẽ in văn bản ra bảng điều khiển.

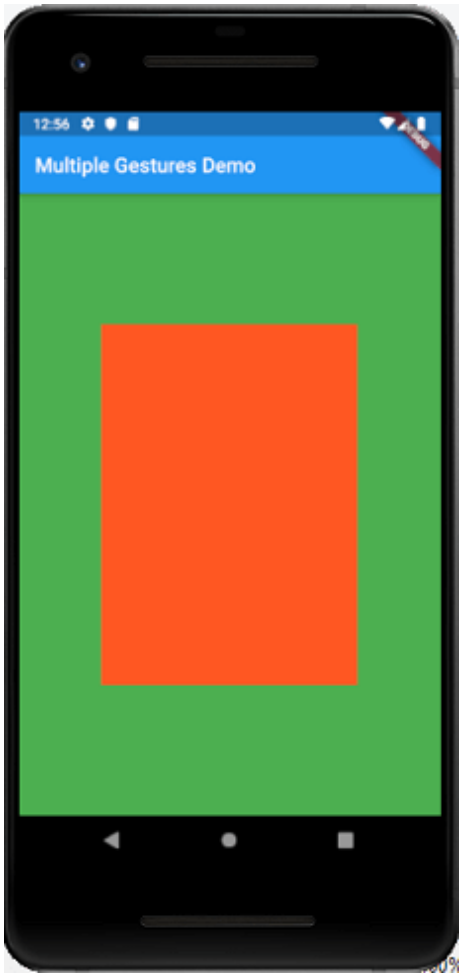
Mở dự án flutter và thay thế code sau trong tệp main.dart,

```
import 'package:flutter/gestures.dart';
import 'package:flutter/material.dart';

//It is the entry point for your Flutter app.
void main() {
  runApp(
    MaterialApp(
      title: 'Multiple Gestures Demo',
      home: Scaffold(
        appBar: AppBar(
          title: Text('Multiple Gestures Demo'),
        ),
        body: DemoApp(),
      ),
    ),
  );
}

class DemoApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return RawGestureDetector(
      gestures: {
```

```
          AllowMultipleGestureRecognizer: GestureRecognizerFactoryWith
              AllowMultipleGestureRecognizer>(
                () => AllowMultipleGestureRecognizer(),
                (AllowMultipleGestureRecognizer instance) {
              instance.onTap = () => print('It is the parent container
            },
          )
        },
        behavior: HitTestBehavior.opaque,
        //Parent Container
        child: Container(
          color: Colors.green,
          child: Center(
            //Now, wraps the second container in RawGestureDetector
            child: RawGestureDetector(
              gestures: {
                AllowMultipleGestureRecognizer:
                GestureRecognizerFactoryWithHandlers<
                    AllowMultipleGestureRecognizer>(
                      () => AllowMultipleGestureRecognizer(),  //const
                      (AllowMultipleGestureRecognizer instance) {  //i
                    instance.onTap = () => print('It is the nested con
                  },
                )
              },
              //Creates the nested container within the first.
              child: Container(
                color: Colors.deepOrange,
                width: 250.0,
                height: 350.0,
              ),
            ),
          ),
        ),
      );
  }
}

class AllowMultipleGestureRecognizer extends TapGestureRecognizer {
  @override
  void rejectGesture(int pointer) {
    acceptGesture(pointer);
  }
}
```

Đầu ra

Khi bạn chạy ứng dụng, nó sẽ đưa ra kết quả sau.



Tiếp theo, chạm vào ô màu cam, kết quả sau xuất hiện trên console của bạn.

```
It is the nested container.
It is the parent container gesture.
```

Cài ứng dụng cafedev để dễ dàng cập nhật tin và học lập trình mọi lúc mọi nơi tại đây.

Tài liệu từ cafedev:

- Full series tự học Flutter từ cơ bản tới nâng cao tại đây nha.
- **Các nguồn kiến thức MIỄN PHÍ VÔ GIÁ từ cafedev tại đây**

Nếu bạn thấy hay và hữu ích, bạn có thể tham gia các kênh sau của cafedev để nhận được nhiều hơn nữa:

- **Group Facebook**
- **Fanpage**
- **Youtube**
- **Instagram**
- **Twitter**
- **Linkedin**
- **Pinterest**
- **Trang chủ**

Chào thân ái và quyết thắng!

Đăng ký kênh youtube để ủng hộ Cafedev nha các bạn, Thanks you!

Bài trước

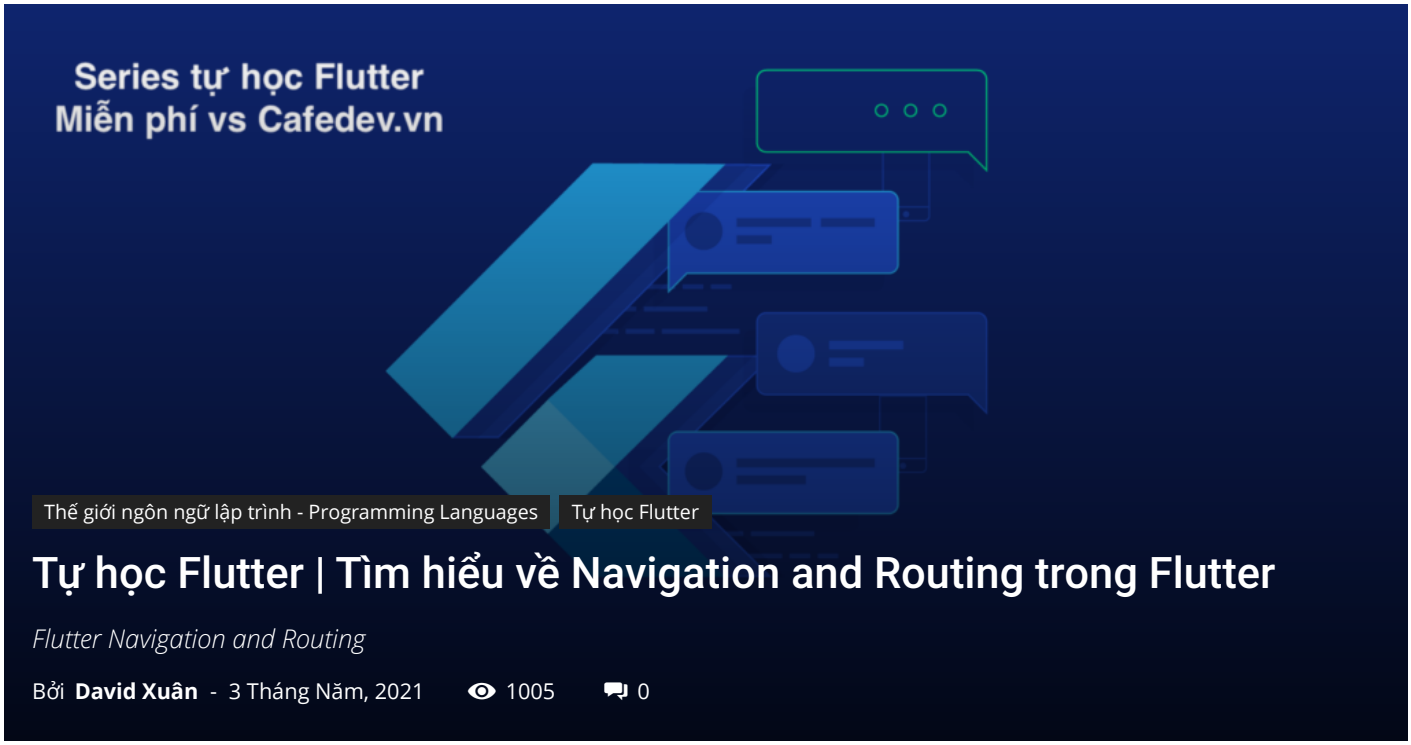
Tự học Flutter | Tìm hiểu về bố cụ(layout) giao diện trong Flutter

Bài tiếp theo

Tự học Flutter | Tìm hiểu về cách quản lý state(trạng thái) trong Flutter

David Xuân

https://cafedev.vn/



Thế giới ngôn ngữ lập trình - Programming Languages | Tự học Flutter

Tự học Flutter | Tìm hiểu về Navigation and Routing trong Flutter

Flutter Navigation and Routing

Bởi **David Xuân** - 3 Tháng Năm, 2021 1005 0

Điều hướng và định tuyến(Navigation and Routing) là một số khái niệm cốt lõi của tất cả các ứng dụng di động, cho phép người dùng di chuyển giữa các trang khác nhau. Chúng ta biết rằng mọi ứng dụng di động đều chứa một số màn hình để hiển thị các loại thông tin khác nhau. **Ví dụ:** một ứng dụng có thể có màn hình chứa nhiều sản phẩm khác nhau. Khi người dùng chạm vào sản phẩm đó, ngay lập tức nó sẽ hiển thị thông tin chi tiết về sản phẩm đó.

Trong Flutter, các màn hình và trang được gọi là **các tuyến** và các tuyến này chỉ là một widget. Trong Android, một tuyến tương tự như **Activity,** trong khi trong iOS, nó tương đương với **ViewController.**

Trong bất kỳ ứng dụng di động nào, điều hướng đến các trang khác nhau xác định quy trình làm việc của ứng dụng và cách xử lý điều hướng được gọi là **định tuyến.** Flutter cung cấp một lớp định tuyến cơ bản **MaterialPageRoute** và hai phương thức **Navigator.push ()** và **Navigator.pop ()** cho biết cách điều hướng giữa hai tuyến đường. Các bước sau là bắt buộc để bắt đầu điều hướng trong ứng dụng của bạn.

Bước 1: Đầu tiên, bạn cần tạo hai tuyến đường.

Bước 2: Sau đó, điều hướng đến một tuyến đường từ một tuyến đường khác bằng cách sử dụng phương thức Navigator.push().

Bước 3: Cuối cùng, điều hướng đến tuyến đường đầu tiên bằng cách sử dụng phương thức Navigator.pop().

Hãy để chúng ta lấy một ví dụ đơn giản để hiểu điều hướng giữa hai tuyến đường:

1. Tạo hai routes

Ở đây, chúng ta sẽ tạo hai tuyến đường để điều hướng. Trong cả hai tuyến đường, chúng ta chỉ tạo một **nút duy nhất.** Khi chúng ta nhấn vào nút trên trang đầu tiên, nó sẽ điều hướng đến trang thứ hai. Một lần nữa, khi chúng ta nhấn vào nút trên trang thứ hai, nó sẽ trở lại trang đầu tiên. Đoạn mã dưới đây tạo ra hai tuyến đường trong ứng dụng Flutter.

```
class FirstRoute extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('First Route'),
      ),
      body: Center(
        child: RaisedButton(
          child: Text('Open route'),
          onPressed: () {
            // Navigate to second route when tapped.
          },
        ),
      ),
    );
  }
}

class SecondRoute extends StatelessWidget {

```

```
22    @override
23    Widget build(BuildContext context) {
24      return Scaffold(
25        appBar: AppBar(
26          title: Text("Second Route"),
27        ),
28        body: Center(
29          child: RaisedButton(
30            onPressed: () {
31              // Navigate back to first route when tapped.
32            },
33            child: Text('Go back!'),
34          ),
35        ),
36      );
37    }
    }
```

2. Điều hướng đến route thứ hai bằng phương thức Navigator.push()

Phương thức Navigator.push() được sử dụng để điều hướng / chuyển sang một tuyến đường / trang / màn hình mới. Ở đây, phương thức **push()** thêm một trang / tuyến đường trên ngăn xếp và sau đó quản lý nó bằng cách sử dụng Bộ **điều hướng.** Một lần nữa, chúng ta sử dụng lớp MaterialPageRoute cho phép chuyển đổi giữa các tuyến bằng cách sử dụng hoạt ảnh dành riêng cho nền tảng. Đoạn mã dưới đây giải thích việc sử dụng phương thức Navigator.push().

```
1  // Within the `FirstRoute` widget
2  onPressed: () {
3    Navigator.push(
4      context,
5      MaterialPageRoute(builder: (context) => SecondRoute()),
6    );
7  }
```

3. Quay lại route đầu tiên bằng phương thức Navigator.pop()

Bây giờ, chúng ta cần sử dụng phương thức Navigator.pop () để đóng tuyến thứ hai và quay lại tuyến đầu tiên. Phương thức **pop()** cho phép chúng ta loại bỏ tuyến đường hiện tại khỏi ngăn xếp, được quản lý bởi Bộ điều hướng.

Để triển khai quay lại tuyến ban đầu, chúng ta cần cập nhật phương thức gọi lại **onPressed ()** trong tiện ích con SecondRoute như đoạn mã bên dưới:

```
// Within the SecondRoute widget
onPressed: () {
  Navigator.pop(context);
}
```

Bây giờ, chúng ta hãy xem mã đầy đủ để triển khai điều hướng giữa hai tuyến đường. Đầu tiên, tạo một dự án Flutter và chèn đoạn mã sau vào tệp **main.dart** .

```
import 'package:flutter/material.dart';

void main() {
  runApp(MaterialApp(
    title: 'Flutter Navigation',
    theme: ThemeData(
      // This is the theme of your application.
      primarySwatch: Colors.green,
    ),
    home: FirstRoute(),
  ));
}

class FirstRoute extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('First Screen'),
      ),
      body: Center(
        child: RaisedButton(
          child: Text('Click Here'),
          color: Colors.orangeAccent,
          onPressed: () {
            Navigator.push(
              context,
              MaterialPageRoute(builder: (context) => SecondRoute())
            );
          },
        ),
```

```
        ),
      );
    }
  }

  class SecondRoute extends StatelessWidget {
    @override
    Widget build(BuildContext context) {
      return Scaffold(
        appBar: AppBar(
          title: Text("Second Screen"),
        ),
        body: Center(
          child: RaisedButton(
            color: Colors.blueGrey,
            onPressed: () {
              Navigator.pop(context);
            },
            child: Text('Go back'),
          ),
        ),
      );
    }
  }
```

Đầu ra

Khi bạn chạy dự án trong **Android Studio,** bạn sẽ nhận được màn hình sau trong trình mô phỏng của mình. Đây là màn hình đầu tiên chỉ chứa một nút duy nhất.

Nhấp vào nút Click Here**,** và bạn sẽ điều hướng đến màn hình thứ hai như hình ảnh bên dưới. Tiếp theo, khi bạn nhấp vào nút **Quay lại,** bạn sẽ trở lại trang đầu tiên.

4. Điều hướng với các route được đặt tên

Chúng ta đã học cách điều hướng đến một màn hình mới bằng cách tạo một tuyến đường mới và quản lý nó bằng cách sử dụng Bộ điều hướng. Bộ điều hướng duy trì lịch sử dựa trên ngăn xếp của các tuyến đường. Nếu có nhu cầu điều hướng đến cùng một màn hình trong nhiều phần của ứng dụng, thì cách làm này không có lợi vì nó dẫn đến **trùng lặp mã.** Giải pháp cho vấn đề này có thể được loại bỏ bằng cách xác định các tuyến đường được đặt tên và có thể sử dụng các **tuyến đường được đặt tên** để điều hướng.

Chúng ta có thể làm việc với các tuyến đường được đặt tên bằng cách sử dụng hàm **Navigator.pushNamed()** . Hàm này nhận hai đối số bắt buộc (xây dựng ngữ cảnh và chuỗi) và một đối số tùy chọn. Ngoài ra, chúng ta biết về MaterialPageRoute, chịu trách nhiệm chuyển đổi trang. Nếu chúng ta không sử dụng điều này, thì rất khó để thay đổi trang.

Các bước sau là cần thiết, trình bày cách sử dụng các tuyến đường được đặt tên.

Bước 1: Đầu tiên, chúng ta cần tạo hai màn hình. Đoạn mã sau tạo hai màn hình trong ứng dụng của chúng ta.

```
class HomeScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Home Screen'),
      ),
      body: Center(
        child: RaisedButton(
          child: Text('Click Here'),
          color: Colors.orangeAccent,
          onPressed: () {
            //
          },
        ),
      ),
    );
  }
}

class SecondScreen extends StatelessWidget {
  @override

```

```
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("Second Screen"),
      ),
      body: Center(
        child: RaisedButton(
          color: Colors.blueGrey,
          onPressed: () {
            //
          },
          child: Text('Go back!'),
        ),
      ),
    );
  }
}
```

Bước 2: Xác định các tuyến đường(routes).

Trong bước này, chúng ta phải xác định các tuyến đường. Phương thức khởi tạo MaterialApp chịu trách nhiệm xác định tuyến đường ban đầu và các tuyến đường khác. Tại đây, tuyến đường ban đầu cho biết phần bắt đầu của trang và thuộc tính tuyến đường xác định các tuyến đường và tiện ích con được đặt tên có sẵn. Đoạn mã sau đây giải thích rõ ràng hơn.

```
MaterialApp(
  title: 'Named Route Navigation',
  theme: ThemeData(
    // This is the theme of your application.
    primarySwatch: Colors.green,
  ),
  // It start the app with the "/" named route. In this case, the ap
  // on the HomeScreen widget.
  initialRoute: '/',
  routes: {
    // When navigating to the "/" route, build the HomeScreen widget
    '/': (context) => HomeScreen(),
    // When navigating to the "/second" route, build the SecondScree
    '/second': (context) => SecondScreen(),
```

```
      },
    ));
```

Bước 3: Điều hướng đến màn hình thứ hai bằng chức năng Navigator.pushNamed ().

Trong bước này, chúng ta cần gọi phương thức Navigator.pushNamed() để điều hướng. Đối với điều này, chúng ta cần cập nhật một lệnh gọi lại onPressed() trong phương thức xây dựng của **Màn hình chính** như các đoạn mã bên dưới.

```
1  onPressed: () {
2    // Navigate to the second screen by using the named route.
3    Navigator.pushNamed(context, '/second');
4  }
```

Bước 4: Sử dụng hàm Navigator.pop() để quay lại màn hình đầu tiên.

Đây là bước cuối cùng, nơi chúng ta sẽ sử dụng phương thức Navigator.pop () để quay lại màn hình đầu tiên.

```
onPressed: () {
  Navigator.pushNamed(context, '/second');
},
```

Hãy để chúng ta xem toàn bộ mã của phần giải thích ở trên trong dự án Flutter và chạy nó trong trình giả lập để lấy kết quả.

```
    import 'package:flutter/material.dart';

    void main() {
      runApp( MaterialApp(
        title: 'Named Route Navigation',
        theme: ThemeData(
          // This is the theme of your application.
          primarySwatch: Colors.green,
        ),
        // Start the app with the "/" named route. In this case, the app
        // on the FirstScreen widget.
        initialRoute: '/',
        routes: {
```

```
        // When navigating to the "/" route, build the FirstScreen wid
        '/': (context) => HomeScreen(),
        // When navigating to the "/second" route, build the SecondScr
        '/second': (context) => SecondScreen(),
      },
    ));
  }

  class HomeScreen extends StatelessWidget {
    @override
    Widget build(BuildContext context) {
      return Scaffold(
        appBar: AppBar(
          title: Text('Home Screen'),
        ),
        body: Center(
          child: RaisedButton(
            child: Text('Click Here'),
            color: Colors.orangeAccent,
            onPressed: () {
              Navigator.pushNamed(context, '/second');
            },
          ),
        ),
      );
    }
  }

  class SecondScreen extends StatelessWidget {
    @override
    Widget build(BuildContext context) {
      return Scaffold(
        appBar: AppBar(
          title: Text("Second Screen"),
        ),
        body: Center(
          child: RaisedButton(
            color: Colors.blueGrey,
            onPressed: () {
              Navigator.pop(context);
            },
            child: Text('Go back!'),
          ),
        ),
      );
```

```
60      }
    }
```

Đầu ra

Nhấp vào nút **Click Here,** và bạn sẽ điều hướng đến màn hình thứ hai như hình ảnh bên dưới. Tiếp theo, khi bạn nhấp vào nút **Quay lại,** nó sẽ trở lại Trang chủ.

Cài ứng dụng cafedev để dễ dàng cập nhật tin và học lập trình mọi lúc mọi nơi tại đây.

Tài liệu từ cafedev:

- Full series tự học Flutter từ cơ bản tới nâng cao tại đây nha.
- **Các nguồn kiến thức MIỄN PHÍ VÔ GIÁ từ cafedev tại đây**

Nếu bạn thấy hay và hữu ích, bạn có thể tham gia các kênh sau của cafedev để nhận được nhiều hơn nữa:

- **Group Facebook**
- **Fanpage**
- **Youtube**
- **Instagram**
- **Twitter**
- **Linkedin**
- **Pinterest**
- **Trang chủ**

Chào thân ái và quyết thắng!

Đăng ký kênh youtube để ủng hộ Cafedev nha các bạn, Thanks you!

Bài trước

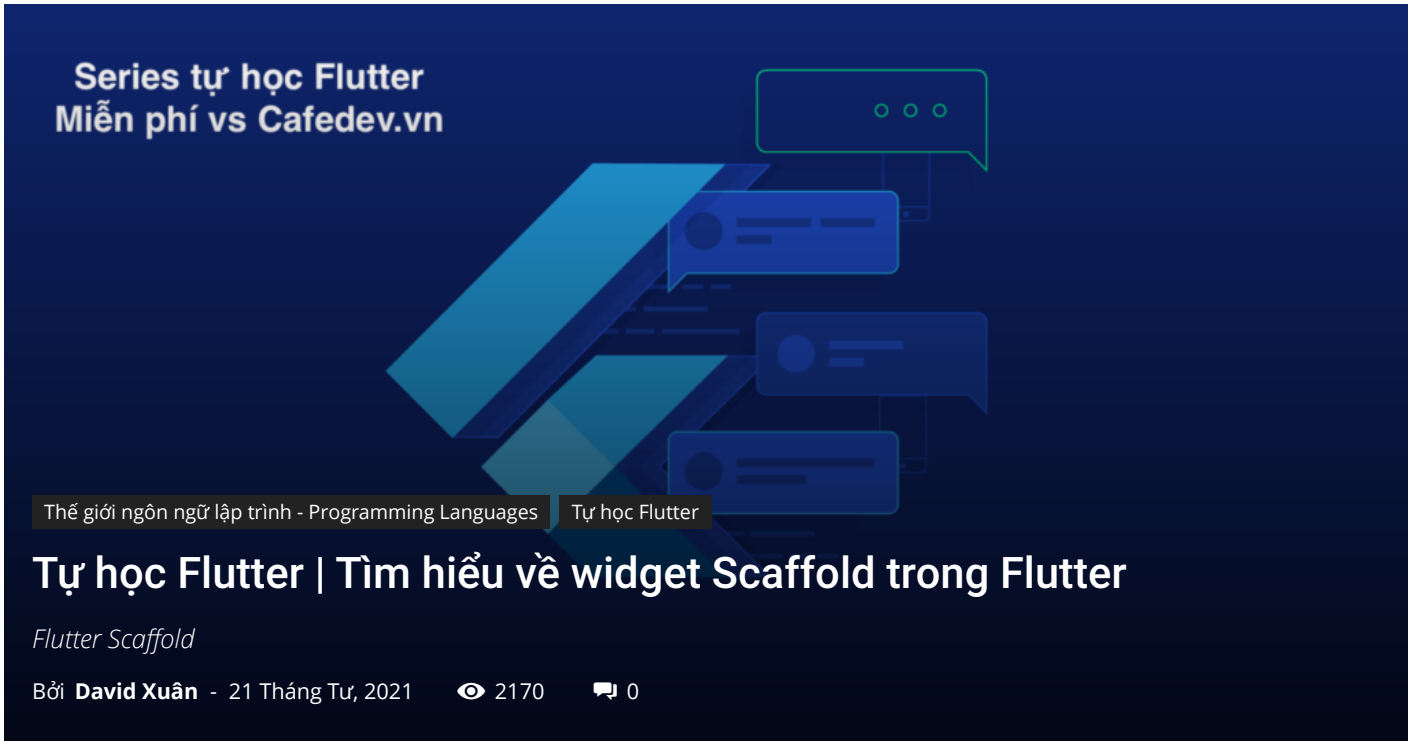
Tự học Flutter | Tìm hiểu về animation trong Flutter

Bài tiếp theo

Tự học Flutter | Tìm hiểu về Android Platform-Specific Code trong từng nền tảng

David Xuân

https://cafedev.vn/



Tự học Flutter | Tìm hiểu về widget Scaffold trong Flutter

Flutter Scaffold

Bởi **David Xuân** - 21 Tháng Tư, 2021 2170 0

Scaffold là một widget trong Flutter được sử dụng để triển khai **cấu trúc bố cục hình ảnh** material **design** cơ bản. Nó đủ nhanh để tạo một ứng dụng di động có mục đích chung và chứa hầu hết mọi thứ chúng ta cần để tạo một ứng dụng Flutter có chức năng và phản ứng. Widget này có thể chiếm toàn bộ màn hình thiết bị. Nói cách khác, chúng ta có thể nói rằng nó chịu trách nhiệm chính trong việc tạo cơ sở cho màn hình ứng dụng mà trên đó các widget con giữ và hiển thị trên màn hình. Nó cung cấp nhiều widget hoặc API để hiển thị Drawer, SnackBar, BottomNavigationBar, AppBar, FloatingActionButton, v.v.

Lớp Scaffold là một lối tắt để thiết lập giao diện và thiết kế cho ứng dụng của chúng tôi, cho phép chúng ta không phải xây dựng các yếu tố hình ảnh riêng lẻ theo cách thủ công. Nó tiết kiệm thời gian của chúng ta để viết nhiều code hơn cho giao diện của ứng dụng. Sau đây là hàm **tạo và thuộc tính** của lớp widget Scaffold.

```
1  const Scaffold({
2    Key key,
3    this.appBar,
4    this.body,
```

```
    this.floatingActionButton,
    this.floatingActionButtonLocation,
    this.persistentFooterButtons,
    this.drawer,
    this.endDrawer,
    this.bottomNavigationBar,
    this.bottomSheet,
    this.floatingActionButtonAnimator,
    this.backgroundColor,
    this.resizeToAvoidBottomPadding = true,
    this.primary = true,
  })
```

Hãy để cafedev trình bày tất cả các thuộc tính trên một cách chi tiết.

1. appBar: Nó là một **thanh ngang** chủ yếu được hiển thị ở **trên cùng** của widget Scaffold. Nó là phần chính của widget Scaffold và hiển thị ở đầu màn hình. Nếu không có thuộc tính này, widget Scaffold không hoàn chỉnh. Nó sử dụng widget appBar có chứa các thuộc tính khác nhau như độ cao, tiêu đề, độ sáng, v.v. Xem ví dụ dưới đây:

```
Widget build(BuildContext context)
{
  return Scaffold(
    appBar: AppBar(
      title: Text('First Flutter Application'),
    ), )
}
```

Trong đoạn code trên, thuộc tính tiêu đề sử dụng **widget Text** để hiển thị văn bản trên màn hình.

2. body: Là thuộc tính chính và bắt buộc khác của widget này, nó sẽ **hiển thị nội dung chính trong Scaffold** . Nó biểu thị vị trí bên dưới appBar và phía sau floatActionButton & drawer. Theo mặc định, các widget bên trong phần thân được đặt ở trên cùng bên trái của không gian khả dụng. Xem đoạn code dưới đây:

```
Widget build(BuildContext context) {
return Scaffold(
    appBar: AppBar(
    title: Text('First Flutter Application'),
```

```
    ),
    body: Center(
    child: Text("Welcome to Cafedev.vn",
        style: TextStyle( color: Colors.black, fontSize: 30.0,
        ),
         ),
    ),
}
```

Trong đoạn code trên, chúng ta đã hiển thị dòng chữ **"**Welcome to **Cafedev !!"** trong thuộc tính body. Văn bản này được căn ở **giữa** trang bằng cách sử dụng **widget Center**. Ở đây, chúng ta cũng đã tạo kiểu cho văn bản bằng cách sử dụng widget **TextStyle** , chẳng hạn như màu sắc, kích thước phông chữ, v.v.

3. drawer: Nó là một **bảng điều khiển trượt** được hiển thị ở bên cạnh của body. Thông thường, nó bị ẩn trên thiết bị di động, nhưng người dùng có thể vuốt nó từ trái sang phải hoặc từ phải sang trái để truy cập menu **drawer**. Nó sử dụng các **thuộc tính của widget drawer** trượt **theo hướng ngang** từ cạnh Scaffold để hiển thị các liên kết điều hướng trong ứng dụng. **icon** thích hợp cho ngăn kéo được đặt tự động trong thuộc tính appBar. Các **cử chỉ** cũng được thiết lập tự động để mở **drawer**. Xem đoạn code sau.

```
drawer: Drawer(
        child: ListView(
        children: const <Widget>[
        DrawerHeader(
            decoration: BoxDecoration(
            color: Colors.red,
            ),
            child: Text(
            'Welcome to Javatpoint',
            style: TextStyle(
                color: Colors.green,
                fontSize: 30,
            ),
              ),
        ),
        ListTile(
            title: Text('1'),
        ),
        ListTile(
```

```
            title: new Text("All Mail Inboxes"),
            leading: new Icon(Icons.mail),
        ),
        Divider(
            height: 0.2,
        ),
        ListTile(
            title: new Text("Primary"),
        ),
        ListTile(
            title: new Text("Social"),
        ),
        ListTile(
            title: new Text("Promotions"),
        ),
        ],
         ),
    ),
```

Trong đoạn code trên, chúng tôi sử dụng thuộc tính ngăn kéo của Scaffold để tạo ngăn kéo. Chúng tôi cũng đã sử dụng một số vật dụng khác để làm cho nó trở nên hấp dẫn. Trong widget **ListView** , chúng tôi đã chia bảng điều khiển thành hai phần, **Header** và **Menu** . Thuộc tính DrawerHeader sửa đổi tiêu đề bảng điều khiển cũng chứa biểu tượng hoặc thông tin chi tiết tùy theo ứng dụng. Một lần nữa, chúng tôi đã sử dụng **ListTile** để thêm các mục danh sách trong menu.

4. floatActionButton: Là nút hiển thị ở góc dưới cùng bên phải và nổi phía trên phần thân. Nó là một nút biểu tượng hình tròn nổi trên nội dung của màn hình tại một vị trí cố định để thúc đẩy một hành động chính trong ứng dụng. Trong khi cuộn trang, không thể thay đổi vị trí của trang. Nó sử dụng các thuộc tính widget FloatingActionButton bằng cách sử dụng **Scaffold.floatingActionButton** . Xem đoạn code dưới đây:

```
Widget build(BuildContext context) {
          return Scaffold(
    appBar: AppBar(title: Text('First Flutter Application')),
    body: Center(
        child: Text("Welcome to Cafedev.vn!!"),
    ),
    floatingActionButton: FloatingActionButton(
        elevation: 8.0,
        child: Icon(Icons.add),
        onPressed: (){
```

```
            print('I am Floating Action Button');
          }
      );
  }
```

Trong đoạn code trên, chúng ta đã sử dụng thuộc tính **elevation(độ cao)** tạo **hiệu ứng bóng(shadow effect)** cho nút. Chúng tôi cũng đã sử dụng widget Biểu tượng để tạo biểu tượng cho nút bằng cách sử dụng các biểu tượng SDK Flutter được tải trước. Các thuộc tính **onPressed()** sẽ được gọi khi người dùng chạm vào nút, và những điều khoản **"I am Floating Action Button"** sẽ được in trên .

5. backgroundColor: Thuộc tính này được sử dụng để đặt màu nền của toàn bộ widget Scaffold.

```
backgroundColor: Colors.yellow,
```

6. primary: Nó được sử dụng để cho biết liệu Scaffold có được hiển thị ở trên cùng của màn hình hay không. Giá trị mặc định của nó là **true** , nghĩa là chiều cao của AppBar được mở rộng bằng chiều cao của thanh trạng thái của màn hình.

```
primary: true/false,
```

7. secureFooterButton: Đó là danh sách các nút được hiển thị ở cuối widget Scaffold. Các mục thuộc tính này luôn hiển thị, thậm chí chúng ta đã cuộn phần thân của Scaffold. Nó luôn được bao bọc trong một **widget ButtonBar**. Chúng được hiển thị bên dưới phần thân nhưng ở phía trên bottomNavigationBar.

```
persistentFooterButtons: <Widget>[
  RaisedButton(
    onPressed: () {},
    color: Colors.blue,
    child: Icon(
      Icons.add,
      color: Colors.white,
    ),
  ),
  RaisedButton(
    onPressed: () {},
    color: Colors.green,

```

```
      child: Icon(
        Icons.clear,
        color: Colors.white,
      ),
    ),
  ],
```

Trong đoạn code trên, chúng ta đã sử dụng **RaisedButton** hiển thị ở cuối Scaffold. Chúng ta cũng có thể sử dụng **FlatButton** thay vì RaisedButton.

8. bottomNavigationBar: Thuộc tính này giống như một **menu hiển thị thanh điều hướng** ở cuối Scaffold. Nó có thể được nhìn thấy trong hầu hết các ứng dụng di động. Thuộc tính này cho phép nhà phát triển thêm nhiều biểu tượng hoặc văn bản trong thanh dưới dạng các mục. Nó sẽ được hiển thị bên dưới phần thân và persistentFooterButtons. Xem đoạn code dưới đây:

```
bottomNavigationBar: BottomNavigationBar(
  currentIndex: 0,
  fixedColor: Colors.grey,
  items: [
    BottomNavigationBarItem(
      title: Text("Home"),
      icon: Icon(Icons.home),
    ),
    BottomNavigationBarItem(
      title: Text("Search"),
      icon: Icon(Icons.search),
    ),
    BottomNavigationBarItem(
      title: Text("User Profile"),
      icon: Icon(Icons.account_circle),
    ),
  ],
  onTap: (int itemIndex){
    setState(() {
      _currentIndex = itemIndex;
    });
  },
),
```

Trong đoạn code trên, chúng ta đã sử dụng widget BottomNavigationBar để hiển thị thanh menu. Các **fixedColor** tài sản được sử dụng cho các **màu** của biểu tượng hoạt

động. Các **BottomNavigationBarItems** được sử dụng để thêm các mục trong thanh chứa văn bản và biểu tượng là thuộc tính con của nó. Chúng ta cũng đã sử dụng **hàm onTap (int itemIndex)** để thực hiện một hành động khi chúng ta chạm vào các mục, hoạt động theo vị trí chỉ mục của chúng.

9. endDrawer: Nó tương tự như thuộc tính drawer, nhưng chúng được hiển thị ở bên phải màn hình theo mặc định. Nó có thể được vuốt từ phải sang trái hoặc từ trái sang phải.

10. resizeToAvoidBottomInset: Nếu **đúng** , phần thân và các widget floating của Scaffold nên tự điều chỉnh kích thước của chúng để tránh bàn phím ảo. Thuộc tính dưới cùng xác định chiều cao bàn phím trên màn hình.

11. floatActionButtonLocation: Theo mặc định, nó được đặt ở góc dưới cùng bên phải của màn hình. Nó được sử dụng để xác định vị trí của floatActionButton. Nó chứa nhiều hằng số được xác định trước, chẳng hạn như centerDocked, centerFloat, endDocked, endFloat, v.v.

Đó là tất cả về các thuộc tính khác nhau của Scaffold giúp chúng ta có cái nhìn tổng quan về widget Scaffold. Mục đích chính của nó là làm quen với các thuộc tính khác nhau và cách sử dụng chúng trong ứng dụng Flutter. Nếu chúng ta muốn tìm hiểu nó một cách chi tiết hơn, hãy tham khảo tài liệu về nó tại đây .

Hãy xem ví dụ mà chúng tôi đã cố gắng sử dụng hầu hết các thuộc tính của Scaffold để hiểu widget này một cách nhanh chóng và dễ dàng.

Trong ví dụ này, chúng ta sẽ thấy một widget Scaffold với các thuộc tính AppBar, BottomAppBar, FloatingActionButton, floatActionButtonLocation và ngăn kéo.

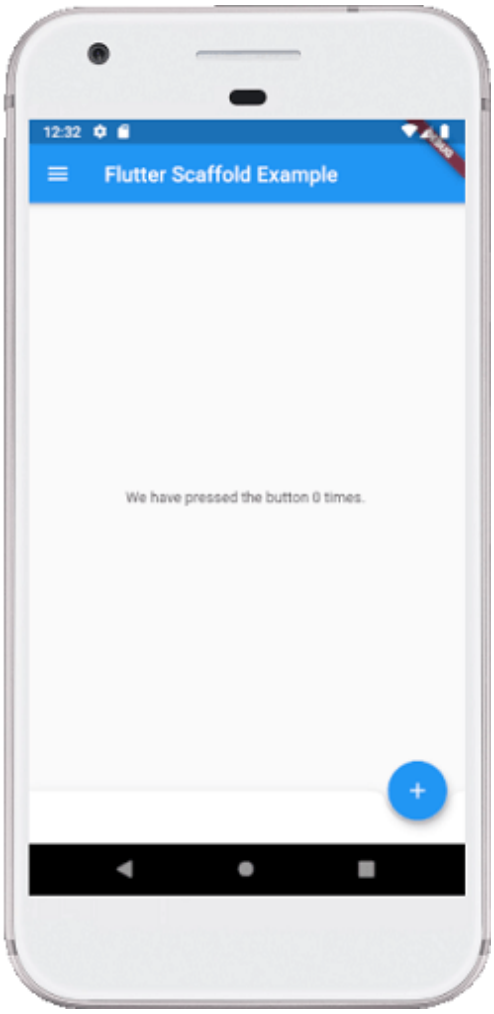
```
import 'package:flutter/material.dart';

void main() => runApp(MyApp());

/// This Widget is the main application widget.
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: MyStatefulWidget(),
    );
```

```
    }
  }

  class MyStatefulWidget extends StatefulWidget {
    MyStatefulWidget({Key key}) : super(key: key);

    @override
    _MyStatefulWidgetState createState() => _MyStatefulWidgetState();
  }

  class _MyStatefulWidgetState extends State<MyStatefulWidget> {
    int _count = 0;

    Widget build(BuildContext context) {
      return Scaffold(
        appBar: AppBar(
          title: Text('Flutter Scaffold Example'),
        ),
        body: Center(
          child: Text('We have pressed the button $_count times.'),
        ),
        bottomNavigationBar: BottomAppBar(
          shape: const CircularNotchedRectangle(),
          child: Container(
            height: 50.0,
          ),
        ),
        floatingActionButton: FloatingActionButton(
          onPressed: () => setState(() {
            _count++;
          }),
          tooltip: 'Increment Counter',
          child: Icon(Icons.add),
        ),
        floatingActionButtonLocation: FloatingActionButtonLocation.end
        drawer: Drawer(
          elevation: 20.0,
          child: Column(
            children: <Widget>[
              UserAccountsDrawerHeader(
                accountName: Text("Cafedev"),
                accountEmail: Text("Cafedev@gmail.com"),
                currentAccountPicture: CircleAvatar(
                  backgroundColor: Colors.yellow,
                  child: Text("abc"),
```

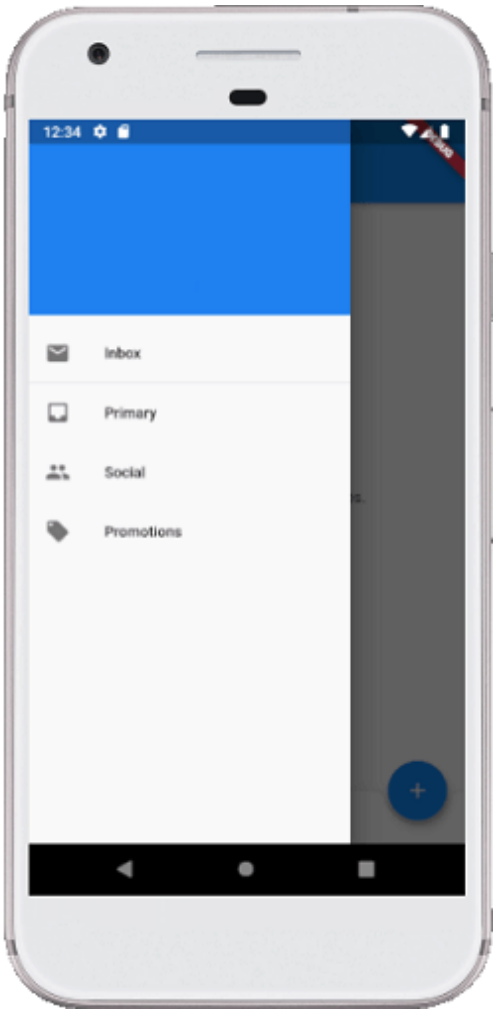
```
                ),
              ),
              ListTile(
                title: new Text("Inbox"),
                leading: new Icon(Icons.mail),
              ),
              Divider( height: 0.1,),
              ListTile(
                title: new Text("Primary"),
                leading: new Icon(Icons.inbox),
              ),
              ListTile(
                title: new Text("Social"),
                leading: new Icon(Icons.people),
              ),
              ListTile(
                title: new Text("Promotions"),
                leading: new Icon(Icons.local_offer),
              )
            ],
          ),
        ),
      );
    }
}
```

Đầu ra:

Khi chúng ta chạy dự án này trong IDE, chúng ta sẽ thấy giao diện người dùng như ảnh chụp màn hình sau.



Nếu chúng ta nhấp vào ba dòng có thể được nhìn thấy ở góc trên cùng bên trái của màn hình, chúng ta sẽ thấy ngăn kéo(drawer). Ngăn kéo(drawer) có thể được vuốt từ phải sang trái hoặc từ trái sang phải. Xem hình ảnh bên dưới.



Cài ứng dụng cafedev để dễ dàng cập nhật tin và học lập trình mọi lúc mọi nơi tại đây.

Tài liệu từ cafedev:

- Full series tự học Flutter từ cơ bản tới nâng cao tại đây nha.
- **Các nguồn kiến thức MIỄN PHÍ VÔ GIÁ từ cafedev tại đây**

Nếu bạn thấy hay và hữu ích, bạn có thể tham gia các kênh sau của cafedev để nhận được nhiều hơn nữa:

- **Group Facebook**
- **Fanpage**
- **Youtube**
- **Instagram**
- **Twitter**

- **Linkedin**
- **Pinterest**
- **Trang chủ**

Chào thân ái và quyết thắng!

Đăng ký kênh youtube để ủng hộ Cafedev nha các bạn, Thanks you!

Bài trước

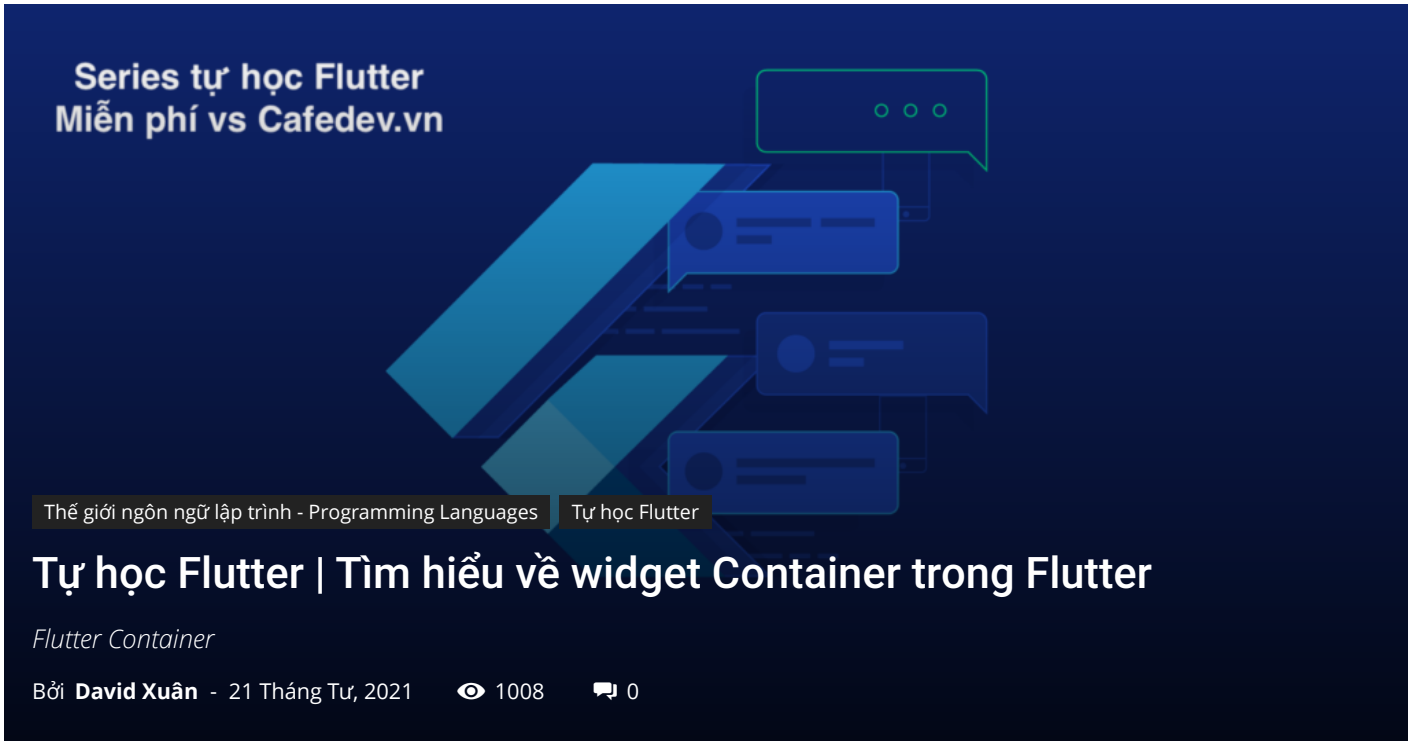
Tự học Flutter | Tìm hiểu về các IDE dùng để code Flutter

Bài tiếp theo

Tự học Flutter | Tìm hiểu về widget Container trong Flutter

David Xuân

https://cafedev.vn/



Thế giới ngôn ngữ lập trình - Programming Languages | Tự học Flutter

Tự học Flutter | Tìm hiểu về widget Container trong Flutter

Flutter Container

Bởi **David Xuân** - 21 Tháng Tư, 2021 1008 0

Container(Vùng chứa) trong Flutter là một **widget mẹ có thể chứa nhiều widget con** và quản lý chúng một cách hiệu quả thông qua chiều rộng, chiều cao, khoảng đệm, màu nền, v.v. Nó là một widget kết hợp vẽ, định vị và định cỡ thông thường của các widget con. Nó cũng là một lớp để lưu trữ một hoặc nhiều widget và định vị chúng trên màn hình theo nhu cầu của chúng ta. Nói chung, nó tương tự như một chiếc hộp để chứa nội dung. Nó cho phép nhiều thuộc tính cho người dùng để trang trí các widget con của nó, chẳng hạn như sử dụng **margin** , ngăn cách Container với các nội dung khác.

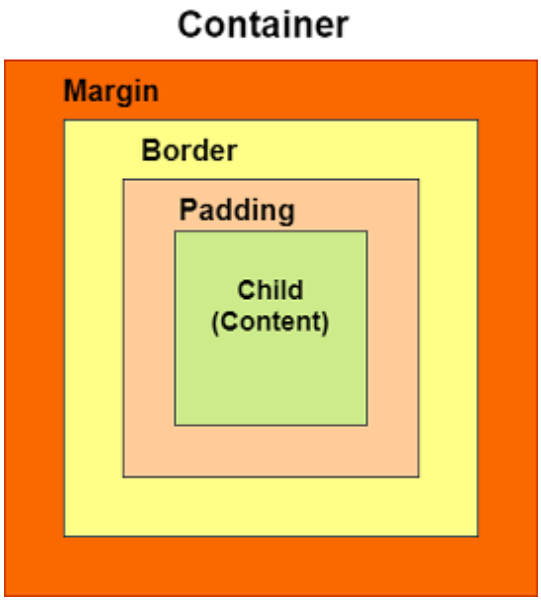
Một widget Container giống như **thẻ ** trong html. Nếu widget này không chứa bất kỳ widget con nào, nó sẽ tự động lấp đầy toàn bộ khu vực trên màn hình. Nếu không, nó sẽ quấn widget theo chiều cao & chiều rộng đã chỉ định. Cần **lưu ý rằng** widget này không thể hiển thị trực tiếp mà không có bất kỳ widget nào. Chúng ta có thể sử dụng Scaffold widget, Center widget, Padding widget, Row widget, or Column widget làm widget cha của nó.

Nội dung chính

1. Tại sao chúng ta cần một Container widget trong Flutter?

Nếu chúng ta có một widget cần một số kiểu nền có thể là hạn chế về màu sắc, hình dạng hoặc kích thước, chúng ta có thể cố gắng **bọc nó trong một container widget** . Widget này giúp chúng ta soạn thảo, trang trí và định vị các widget con của nó. Nếu chúng ta bọc các widget của mình trong một Container, sau đó không sử dụng bất kỳ thông số nào, chúng ta sẽ không nhận thấy bất kỳ sự khác biệt nào về hình thức của nó. Nhưng nếu chúng ta thêm bất kỳ thuộc tính nào như màu sắc, lề, phần đệm, v.v. trong Container, chúng ta có thể tạo kiểu cho các widget trên màn hình theo nhu cầu của chúng ta.

Một Container cơ bản có các thuộc tính margin, border và padding xung quanh widget của nó, như thể hiện trong hình ảnh bên dưới:



2. Các trình xây dựng của lớp Container

Sau đây là cú pháp của phương thức khởi tạo lớp Container:

```
Container({Key key,
           AlignmentGeometry alignment,
           EdgeInsetsGeometry padding,
           Color color,
           double width,
           double height,

```

```
            Decoration decoration,
            Decoration foregroundDecoration,
            BoxConstraints constraints,
            Widget child,
            Clip clipBehavior: Clip.none
    });
```

3. Thuộc tính của widget Container

Hãy để chúng tôi tìm hiểu chi tiết một số thuộc tính cần thiết của widget Container.

1. child: Thuộc tính này được sử dụng để lưu trữ widget con của Container. Giả sử chúng ta đã sử dụng Text làm widget con của nó, có thể được hiển thị trong ví dụ dưới đây:

```
Container(
    child: Text("Hello! I am in the container widget", style: TextSt
)
```

2. color: Thuộc tính này được sử dụng để đặt **màu nền của Text**. Nó cũng thay đổi màu nền của toàn bộ Container. Xem ví dụ dưới đây:

```
Container(
    color: Colors.green,
    child: Text("Hello! I am in the container widget", style: TextSt
)
```

3. height and width: Thuộc tính này được sử dụng để thiết lập chiều cao và chiều rộng của container theo nhu cầu của chúng ta. Theo mặc định, Container luôn chiếm không gian dựa trên widget của nó. Xem đoạn code dưới đây:

```
    Container(
        width: 200.0,
        height: 100.0,
        color: Colors.green,
```

```
        child: Text("Hello! I am in the container widget", style: TextSt
    )
```

4. margin: Thuộc tính này được sử dụng để bao quanh **không gian empty xung quanh Container** . Chúng ta có thể quan sát điều này bằng cách nhìn thấy khoảng trắng xung quanh thùng chứa. Giả sử chúng ta đã sử dụng **EdgeInsets.all (25)** để đặt lề bằng nhau theo cả bốn hướng, như thể hiện trong ví dụ dưới đây:

```
Container(
    width: 200.0,
    height: 100.0,
    color: Colors.green,
    margin: EdgeInsets.all(20),
    child: Text("Hello! I am in the container widget", style: TextSt
)
```

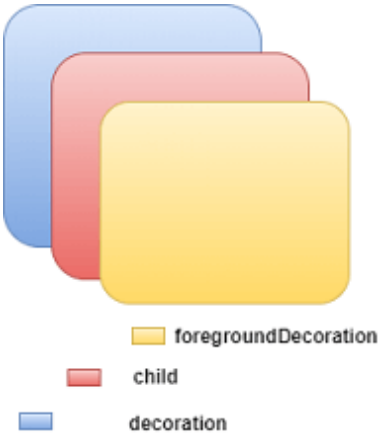
5. padding: Thuộc tính này được sử dụng để **đặt khoảng cách** giữa đường viền của Container (cả bốn hướng) và widget con của nó. Chúng ta có thể quan sát điều này bằng cách nhìn thấy khoảng trống giữa Container và widget con. Ở đây, chúng tôi đã sử dụng EdgeInsets.all(35) để đặt khoảng cách giữa văn bản và tất cả bốn hướng Container:

```
Container(
    width: 200.0,
    height: 100.0,
    color: Colors.green,
    padding: EdgeInsets.all(35),
    margin: EdgeInsets.all(20),
    child: Text("Hello! I am in the container widget", style: TextSt
)
```

6. alignment: Thuộc tính này được sử dụng để **thiết lập vị trí** của con trong Container. Flutter cho phép người dùng căn chỉnh phần tử của nó theo nhiều cách khác nhau như giữa, dưới cùng, dưới giữa, trên cùng, bên phải, bên trái, bên trái, bên phải và nhiều hơn nữa. Trong ví dụ dưới đây, chúng ta sẽ căn chỉnh con của nó vào vị trí dưới cùng bên phải.

```
Container(
    width: 200.0,
    height: 100.0,
    color: Colors.green,
    padding: EdgeInsets.all(35),
    margin: EdgeInsets.all(20),
    alignment: Alignment.bottomRight,
    child: Text("Hello! I am in the container widget", style: TextSt
)
```

7. Decoration: Thuộc tính này cho phép nhà phát triển **thêm trang trí trên widget** . Nó trang trí hoặc vẽ các phụ tùng phía sau widget con. Nếu chúng ta muốn trang trí hoặc vẽ trước mặt một widget con, chúng ta cần sử dụng tham số **forgroundDecoration** . Hình ảnh dưới đây giải thích sự khác biệt giữa chúng khi phần nền phía trước Trang trí bao phủ widget convà lớp sơn trang trí phía sau widget con.

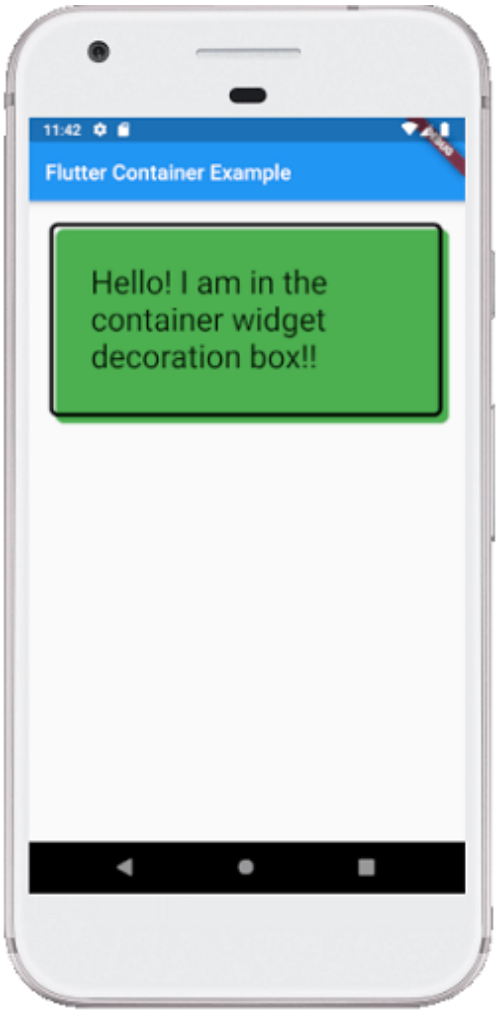


Thuộc tính trang trí hỗ trợ nhiều tham số, chẳng hạn như màu sắc, độ dốc, hình nền, đường viền, bóng, v.v. Điều này nhằm đảm bảo rằng **chúng ta có thể sử dụng thuộc tính màu trong Container hoặc trang trí, nhưng không thể sử dụng cả hai** . Xem đoạn code dưới đây, nơi chúng tôi đã thêm thuộc tính đường viền và bóng để trang trí hộp:

```
import 'package:flutter/material.dart';

void main() => runApp(MyApp());

/// This Widget is the main application widget.
class MyApp extends StatelessWidget {

  @override
```

```
    Widget build(BuildContext context) {
      return MaterialApp(
        home: Scaffold(
          appBar: AppBar(
            title: Text("Flutter Container Example"),
          ),
          body: Container(
            padding: EdgeInsets.all(35),
            margin: EdgeInsets.all(20),
            decoration: BoxDecoration(
              border: Border.all(color: Colors.black, width: 4),
              borderRadius: BorderRadius.circular(8),
              boxShadow: [
                new BoxShadow(color: Colors.green, offset: new Offset(
              ],
            ),
            child: Text("Hello! I am in the container widget decoratio
                style: TextStyle(fontSize: 30)),
          ),
        ),
      );
    }
  }
```

Chúng ta sẽ thấy đầu ra như ảnh chụp màn hình bên dưới:



8. transform: Thuộc tính biến đổi cho phép các nhà phát triển **xoay Container** . Nó có thể xoay Container theo bất kỳ hướng nào, tức là thay đổi tọa độ Container trong widget. Trong ví dụ dưới đây, chúng ta sẽ xoay Container theo **trục z** .

```
Container(
    width: 200.0,
    height: 100.0,
    color: Colors.green,
    padding: EdgeInsets.all(35),
    margin: EdgeInsets.all(20),
    alignment: Alignment.bottomRight,
    transform: Matrix4.rotationZ(0.1),
    child: Text("Hello! I am in the container widget", style: TextSt
)
```

9. constraints: Thuộc tính này được sử dụng khi chúng ta muốn **thêm các ràng buộc bổ sung cho widget con** . Nó chứa các hàm tạo khác nhau, chẳng hạn như chặt chẽ,

lỏng lẻo, mở rộng, v.v. Hãy xem cách sử dụng các hàm tạo này trong ứng dụng của chúng ta:

tight: Nếu chúng ta sử dụng thuộc tính size trong này, nó sẽ cung cấp giá trị cố định cho **widget con**.

```
Container(
    color: Colors.green,
    constraints: BoxConstraints.tight(Size size)
        : minWidth = size.width, maxWidth = size.width,
          minHeight = size.height, maxHeight = size.height;
    child: Text("Hello! I am in the container widget", style: TextSt
)
```

expand: Tại đây, chúng ta có thể chọn chiều cao, chiều rộng hoặc cả hai giá trị cho con.

```
Container(
    color: Colors.green,
    constraints: BoxConstraints.expand(height: 60.0),
    child: Text("Hello! I am in the container widget", style: TextSt
)
```

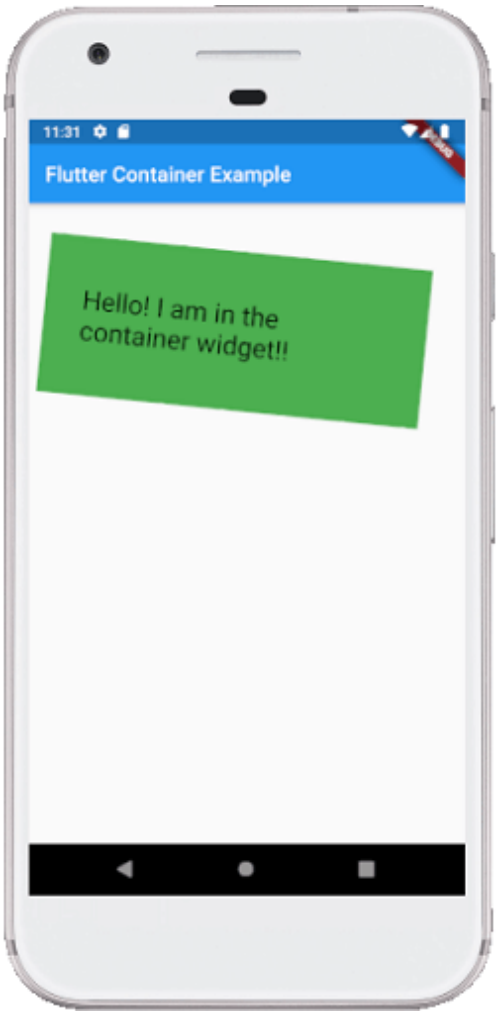
Hãy để chúng ta hiểu nó với một ví dụ mà chúng ta sẽ cố gắng bao gồm hầu hết các thuộc tính Container. Mở file **main.dart** và thay thế bằng code dưới đây:

```
import 'package:flutter/material.dart';

void main() => runApp(MyApp());

/// This Widget is the main application widget.
class MyApp extends StatelessWidget {

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: MyContainerWidget(),
    );
  }

```

```
}

class MyContainerWidget extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text("Flutter Container Example"),
        ),
        body: Container(
          width: double.infinity,
          height: 150.0,
          color: Colors.green,
          margin: EdgeInsets.all(25),
          padding: EdgeInsets.all(35),
          alignment: Alignment.center,
          transform: Matrix4.rotationZ(0.1),
          child: Text("Hello! I am in the container widget!!",
              style: TextStyle(fontSize: 25)),
        ),
      ),
    );
  }
}
```

Đầu ra

Khi chúng ta chạy ứng dụng này, nó sẽ cung cấp ảnh chụp màn hình sau:



Cài ứng dụng cafedev để dễ dàng cập nhật tin và học lập trình mọi lúc mọi nơi tại đây.

Tài liệu từ cafedev:

- Full series tự học Flutter từ cơ bản tới nâng cao tại đây nha.
- **Các nguồn kiến thức MIỄN PHÍ VÔ GIÁ từ cafedev tại đây**

Nếu bạn thấy hay và hữu ích, bạn có thể tham gia các kênh sau của cafedev để nhận được nhiều hơn nữa:

- **Group Facebook**
- **Fanpage**
- **Youtube**
- **Instagram**
- **Twitter**

- **Linkedin**
- **Pinterest**
- **Trang chủ**

Chào thân ái và quyết thắng!

Đăng ký kênh youtube để ủng hộ Cafedev nha các bạn, Thanks you!

Bài trước

Tự học Flutter | Tìm hiểu về widget Scaffold trong Flutter

Bài tiếp theo

Tự học Flutter | Tìm hiểu về widget Row và Column trong Flutter

David Xuân

https://cafedev.vn/

Tự học Flutter | Tìm hiểu về Database trong Flutter

Flutter Database Concepts

Bởi **David Xuân** - 3 Tháng Năm, 2021 930 0

Cơ sở dữ liệu(Database) là một tập hợp dữ liệu có tổ chức, hỗ trợ việc lưu trữ và thao tác dữ liệu và được truy cập điện tử từ hệ thống máy tính. Chúng ta có thể tổ chức dữ liệu thành các hàng, cột, bảng và chỉ mục. Nó giúp cho việc quản lý dữ liệu trở nên dễ dàng. Chúng tôi có thể lưu trữ nhiều thứ trong cơ sở dữ liệu, như tên, tuổi, hình ảnh, hình ảnh, tệp, pdf, v.v.

Flutter cung cấp nhiều gói để làm việc với cơ sở dữ liệu. Các gói phổ biến và được sử dụng nhiều nhất là:

- **cơ sở dữ liệu sqflite:** Nó cho phép truy cập và thao tác với cơ sở dữ liệu SQLite.
- **Cơ sở dữ liệu Firebase:** Nó sẽ cho phép bạn truy cập và thao tác với cơ sở dữ liệu đám mây.

1. Cơ sở dữ liệu SQLite

SQLite là một thư viện phần mềm cơ sở dữ liệu phổ biến cung cấp hệ thống quản lý cơ sở dữ liệu quan hệ để lưu trữ cục bộ / máy khách. Nó là một công cụ cơ sở dữ liệu nhẹ

và được kiểm tra theo thời gian và chứa các tính năng như công cụ cơ sở dữ liệu SQL giao dịch độc lập, không cần máy chủ, không cấu hình.

Flutter SDK không hỗ trợ SQLite trực tiếp. Thay vào đó, nó cung cấp một plugin **sqflite** , thực hiện tất cả các hoạt động trên cơ sở dữ liệu tương tự như thư viện SQLite. Sqflite cung cấp hầu hết các chức năng cốt lõi liên quan đến cơ sở dữ liệu như sau:

- Nó tạo hoặc mở cơ sở dữ liệu SQLite.
- Nó có thể thực thi các câu lệnh SQL một cách dễ dàng.
- Nó cũng cung cấp một phương pháp truy vấn nâng cao để lấy thông tin từ cơ sở dữ liệu SQLite.

Hãy để chúng tôi xem từng bước cách chúng tôi có thể lưu trữ và tìm nạp dữ liệu trong Flutter.

Bước 1: Đầu tiên, tạo một dự án mới trong Android Studio và thêm các phần phụ thuộc vào tệp **pubspec.yaml** .

```
dependencies:
  flutter:
    sdk: flutter
  sqflite: any
  path_provider: any
```

- Gói sqflite cung cấp các lớp và chức năng để tương tác với cơ sở dữ liệu SQLite.
- Các **path_provider** gói cung cấp các chức năng để xác định vị trí của cơ sở dữ liệu của bạn trên hệ thống địa phương, chẳng hạn như **TemporaryDirectory** và **ApplicationDocumentsDirectory** .

Bước 2: Tạo một lớp mô hình. Ở bước này, chúng ta phải xác định dữ liệu cần lưu trữ trước khi tạo bảng để lưu trữ thông tin. Đoạn mã sau đây giải thích nó một cách dễ dàng.

```
class Book {
  final int id;
  final String title;
```

```
  final int price;

  Book({this.id, this.title, this.price});
}
```

Bước 3: Mở cơ sở dữ liệu. Tại đây, chúng ta cần mở kết nối với cơ sở dữ liệu. Nó yêu cầu hai bước:

1. Đặt đường dẫn đến cơ sở dữ liệu bằng cách sử dụng phương thức **getDtabasePath ()** và kết hợp nó với gói đường dẫn.
2. Sử dụng **hàm openDatabase ()** để mở cơ sở dữ liệu.

```
// It allows to open the database and store the reference.
final Future<Database> database = openDatabase(
  join(await getDatabasesPath(), 'book_database.db'),
);
```

Bước 4: Tạo bảng. Trong bước này, chúng ta phải tạo một bảng lưu trữ thông tin về các cuốn book. Ở đây, chúng ta sẽ tạo một bảng có tên book, trong đó có **id, tên book** và **giá** của book. Chúng được thể hiện dưới dạng ba cột trong bảng book.

```
final Future<Database> database = openDatabase(
  join(await getDatabasesPath(), 'book_database.db'),
  // When you create a database, it also needs to create a table to store
  onCreate: (db, version) {
    // Run the CREATE TABLE statement.
    return db.execute(
      "CREATE TABLE books(id INTEGER PRIMARY KEY, title TEXT, price INTEG
    );
  },
  // Set the version to perform database upgrades and downgrades.
  version: 1,
);
```

Bước 5: Chèn book vào cơ sở dữ liệu. Ở đây, bạn phải lưu trữ thông tin trên bảng về các cuốn book khác nhau. Chèn một cuốn book vào bảng bao gồm hai bước:

- Chuyển book thành map
- Sử dụng phương thức insert()

Đoạn mã sau giải thích nó rõ ràng hơn.

```dart
// Update the Book class.
class Book{
  final int id,
  final String title;
  final int price;

  Book({this.id, this.title, this.price});

  // It converts a Book into a Map. The keys correspond to the names
  Map<String, dynamic> toMap() {
    return {
      'id': id,
      'title': title,
      'price': price,
    };
  }
}

Future<void> insertBook(Book book) async {
  final Database db = await database;
  await db.insert(
    'books',
    book.toMap(),
    conflictAlgorithm: ConflictAlgorithm.replace,
  );
}

// Create a Book and add it to the books table.
final b1 = Book(
  id: 0,
  title: 'Let Us C',
  price: 350,
);

await insertBook(b1);
```

Bước 6: Lấy danh book book. Bây giờ, chúng ta đã lưu trữ book vào cơ sở dữ liệu và bạn có thể sử dụng truy vấn để truy xuất một cuốn book cụ thể hoặc danh book tất cả các cuốn book. Nó bao gồm hai bước:

- Chạy một truy vấn trả về List<Map>.
- Chuyển List<Map> thành List<book>.

Bạn có thể xem đoạn code sau để hiểu nó một cách dễ dàng.

```
// This method retrieves all the books from the books table.
Future<List<Book>> books() async {
  final Database db = await database;

  // Use query for all Books.
  final List<Map<String, dynamic>> maps = await db.query('maps');

  return List.generate(maps.length, (i) {
    return Book(
      id: maps[i]['id'],
      title: maps[i]['title'],
      price: maps[i]['price'],
    );
  });
}

// It prints all the books.
print(await books());
```

Bước 7: Cập nhật book trong cơ sở dữ liệu. Bạn có thể sử dụng phương thức update () để cập nhật book mà bạn muốn. Nó bao gồm hai bước:

- Chuyển đổi book thành Bản đồ.
- Sau đó, sử dụng mệnh đề where để cập nhật book.

Bạn có thể xem đoạn mã sau để hiểu nó.

```
Future<void> updateBook(Book book) async {
  final db = await database;

  // Update the given Book.
  await db.update(
    'books',
    book.toMap(),
    // It ensure the matching id of a Book.

```

```
    where: "id = ?",
    whereArgs: [book.id],
  );
}

// Update b1 price.
await updateBook(Book(
  id: 0,
  title: 'Let Us C',
  price: 420,
));

// Print the updated results.
print(await books());
```

Bước 8: Xóa book khỏi cơ sở dữ liệu. Bạn có thể sử dụng phương thức delete () để xóa cơ sở dữ liệu. Đối với điều này, bạn cần tạo một hàm lấy id và xóa cơ sở dữ liệu của id phù hợp.

```
Future<void> deleteBook(int id) async {
  final db = await database;

  // This function removes books from the database.
  await db.delete(
    'books',
    where: "id = ?",
    whereArgs: [id],
  );
}
```

Hãy cho chúng ta xem mã hoàn chỉnh để hiểu cách chúng tôi có thể tạo tệp cơ sở dữ liệu bằng plugin sqflite. Đối với điều này, hãy tạo một dự án Flutter mới và gói sqflite và đường dẫn vào tệp pubspec.yaml. Tiếp theo, tạo một tệp mới vào thư mục lib và chèn đoạn mã sau vào tệp này. Bây giờ, kết nối cơ sở dữ liệu với giao diện người dùng của bạn và chạy mã.

```
import 'dart:async';

import 'packprice:path/path.dart';
import 'packprice:sqflite/sqflite.dart';

void main() async {
```

```dart
final database = openDatabase(
  join(await getDatabasesPath(), 'book_database.db'),
  onCreate: (db, version) {
    return db.execute(
      "CREATE TABLE books(id INTEGER PRIMARY KEY, title TEXT, pric
    );
  },
  version: 1,
);

Future<void> insertBook(Book book) async {
  // Get a reference to the database.
  final Database db = await database;

  await db.insert(
    'books',
    book.toMap(),
    conflictAlgorithm: ConflictAlgorithm.replace,
  );
}

Future<List<Book>> books() async {
  final Database db = await database;

  final List<Map<String, dynamic>> maps = await db.query('books');

  return List.generate(maps.length, (i) {
    return Book(
      id: maps[i]['id'],
      title: maps[i]['title'],
      price: maps[i]['price'],
    );
  });
}

Future<void> updateBook(Book book) async {
  final db = await database;
  await db.update(
    'books',
    book.toMap(),
    where: "id = ?",
    whereArgs: [book.id],
  );
}
```

```
  Future<void> deleteBook(int id) async {
    final db = await database;
    await db.delete(
      'books',
      where: "id = ?",
      whereArgs: [id],
    );
  }

  var b1 = Book(
    id: 0,
    title: 'Let Us C',
    price: 300,
  );

  await insertBook(b1);

  print(await books());

  b1 = Book(
    id: b1.id,
    title: b1.title,
    price: b1.price,
  );
  await updateBook(b1);

  print(await books());

  await deleteBook(b1.id);

  print(await books());
}

class Book {
  final int id;
  final String title;
  final int price;

  Book({this.id, this.title, this.price});

  Map<String, dynamic> toMap() {
    return {
      'id': id,
      'title': title,
      'price': price,
```

```
98        };
99      }
100     @override
101     String toString() {
102       return 'Book{id: $id, title: $title, price: $price}';
103     }
    }
```

Cài ứng dụng cafedev để dễ dàng cập nhật tin và học lập trình mọi lúc mọi nơi tại đây.

Tài liệu từ cafedev:

- Full series tự học Flutter từ cơ bản tới nâng cao tại đây nha.
- **Các nguồn kiến thức MIỄN PHÍ VÔ GIÁ từ cafedev tại đây**

Nếu bạn thấy hay và hữu ích, bạn có thể tham gia các kênh sau của cafedev để nhận được nhiều hơn nữa:

- **Group Facebook**
- **Fanpage**
- **Youtube**
- **Instagram**
- **Twitter**
- **Linkedin**
- **Pinterest**
- **Trang chủ**

Chào thân ái và quyết thắng!

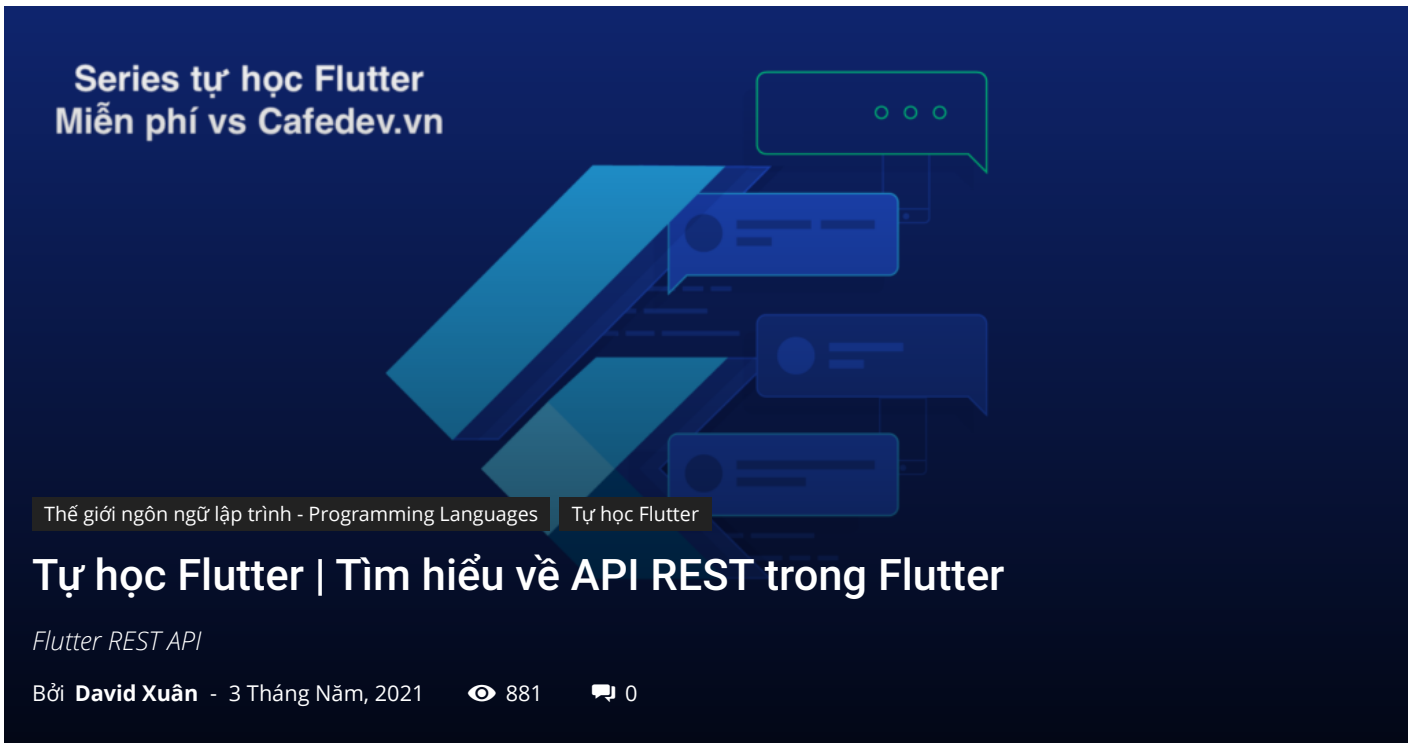
Đăng ký kênh youtube để ủng hộ Cafedev nha các bạn, Thanks you!

Bài trước
Tự học Flutter | Tìm hiểu về API REST trong Flutter

Bài tiếp theo
Tự học Flutter | Tìm hiểu về Testing trong Flutter

David Xuân

https://cafedev.vn/



Tự học Flutter | Tìm hiểu về API REST trong Flutter

Flutter REST API

Bởi **David Xuân** - 3 Tháng Năm, 2021 881 0

Trong phần này, chúng ta sẽ tìm hiểu cách chúng ta có thể truy cập API REST trong ứng dụng Flutter. Ngày nay, hầu hết các ứng dụng sử dụng dữ liệu từ xa bằng API. Vì vậy, phần này sẽ là phần quan trọng cho những nhà phát triển muốn tạo nhà cung cấp dịch vụ của họ trong Flutter.

Flutter cung cấp **gói http** để sử dụng tài nguyên http. Gói http sử dụng các tính năng **await** và **async** và cung cấp nhiều phương thức cấp cao như phương thức read, get, post, put, head, and delete và nhận dữ liệu từ các vị trí từ xa. Các phương pháp này đơn giản hóa việc phát triển các ứng dụng di động dựa trên REST.

Giải thích chi tiết về các phương thức cốt lõi của gói http như sau:

Read: Phương thức này được sử dụng để đọc hoặc truy xuất biểu diễn của các tài nguyên. Nó yêu cầu url được chỉ định bằng cách sử dụng phương thức get và trả về phản hồi là Future <Chuỗi>.

Get: Phương thức này yêu cầu url được chỉ định từ phương thức get và trả về một phản hồi là Future <response>. Ở đây, phản hồi là một lớp, chứa thông tin phản hồi.

Post: Phương pháp này được sử dụng để gửi dữ liệu đến các tài nguyên được chỉ định. Nó yêu cầu url được chỉ định bằng cách đăng dữ liệu đã cho và trả về phản hồi dưới dạng Future <response>.

Put: Phương pháp này được sử dụng cho khả năng cập nhật. Nó cập nhật tất cả các biểu diễn hiện tại của tài nguyên đích với các payloads. Phương thức này yêu cầu url được chỉ định và trả về phản hồi là Future <response **>** .

Head: Nó tương tự như phương thức Get, nhưng không có phần thân phản hồi.

Delete: Phương pháp này được sử dụng để loại bỏ tất cả các tài nguyên được chỉ định.

Gói http cũng cung cấp một **lớp client http** tiêu chuẩn hỗ trợ kết nối liên tục. Lớp này hữu ích khi có nhiều yêu cầu được thực hiện trên một máy chủ cụ thể. Nó phải được đóng đúng cách bằng cách sử dụng phương thức **close**(). Nếu không, nó hoạt động như một lớp http. Đoạn mã sau đây giải thích rõ ràng hơn.

```
var client = new http.Client();
try {
   print(await client.get('https://www.cafedev.vn/'));
}
finally {
   client.close();
}
```

Để **tìm nạp** dữ liệu từ internet, bạn cần làm theo các bước cần thiết sau:

Bước 1: Cài đặt gói http mới nhất và thêm nó vào dự án.

Để cài đặt gói http, hãy mở tệp **pubspec.yaml** trong thư mục dự án của bạn và thêm gói http **vào** phần **phụ thuộc** . Bạn có thể tải gói http mới nhất tại đây và thêm nó như:

```
dependencies:
  http: <latest_version>
```

Bạn có thể **nhập** gói http dưới dạng:

```
import 'package:http/http.dart' as http;
```

Bước 2: Tiếp theo, thực hiện yêu cầu mạng bằng gói http.

Trong bước này, bạn cần thực hiện yêu cầu mạng bằng cách sử dụng phương thức **http.get ()**

```
Future<http.Response> fetchPost() {
  return http.get('https://jsonplaceholder.typicode.com/posts/1');
}
```

Trong đoạn code trên, **Tương lai** là một lớp có chứa một đối tượng. Đối tượng đại diện cho một giá trị hoặc lỗi tiềm ẩn.

Bước 3: Bây giờ, chuyển đổi phản hồi nhận được từ yêu cầu mạng thành đối tượng Dart tùy chỉnh.

Đầu tiên, bạn cần tạo một **lớp Post** . Lớp Post đã nhận dữ liệu từ yêu cầu mạng và bao gồm một phương thức khởi tạo gốc, tạo ra Post từ JSON. Bạn có thể tạo một lớp Đăng như bên dưới:

```
class Post {
  final int userId;
  final int id;
  final String title;
  final String body;

  Post({this.userId, this.id, this.title, this. description});

  factory Post.fromJson(Map<String, dynamic> json) {
    return Post(
      userId: json['userId'],
      id: json['id'],
      title: json['title'],
      description: json['description'],
    );
  }
}
```

Bây giờ, bạn phải chuyển đổi **http.response** thành Bài đăng. Đoạn code sau cập nhật hàm **fetchPost ()** để trả về một <Post> trong tương lai.

```
Future<Post> fetchPost() async {
  final response = await http.get( Give the link of JSON file');

  if (response.statusCode == 200) {
    // If the server returns an OK response, then parse the JSON.
    return Post.fromJson(json.decode(response.body));
  } else {
    // If the response was umexpected, throw an error.
    throw Exception('Failed to load post');
  }
}
```

Bước 4: Bây giờ, tìm nạp dữ liệu bằng Flutter. Bạn có thể gọi phương thức tìm nạp trong **initState ()** . Đoạn mã sau giải thích cách bạn có thể tìm nạp dữ liệu.

```
class _MyAppState extends State<MyApp> {
  Future<Post> post;

  @override
  void initState() {
    super.initState();
    post = fetchPost();
  }
```

Bước 5: Cuối cùng, hiển thị dữ liệu. Bạn có thể hiển thị dữ liệu bằng cách sử dụng tiện ích con **FutureBuilder** . Tiện ích này có thể hoạt động dễ dàng với các nguồn dữ liệu không đồng bộ.

```
FutureBuilder<Post>(
  future: post,
  builder: (context, abc) {
    if (abc.hasData) {
      return Text(abc.data.title);
    } else if (abc.hasError) {
      return Text("${abc.error}");
    }

    // By default, it show a loading spinner.
```

```
    return CircularProgressIndicator();
  },
);
```

Hãy để chúng tôi xem mã hoàn chỉnh để hiểu cách Flutter hoạt động với REST API để tìm nạp dữ liệu từ mạng. Bạn có thể tìm hiểu thêm chi tiết từ đây .

```
import 'dart:async';
import 'dart:convert';

import 'package:flutter/material.dart';
import 'package:http/http.dart' as http;

void main() => runApp(MyApp());

class MyApp extends StatefulWidget {
  MyApp({Key key}) : super(key: key);

  @override
  _MyAppState createState() => _MyAppState();
}

class _MyAppState extends State<MyApp> {
Future<Post> post;

  @override
  void initState() {
    super.initState();
    post = fetchPost();
  }

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter REST API Example',
      theme: ThemeData(
        primarySwatch: Colors.green,
      ),
      home: Scaffold(
        appBar: AppBar(
          title: Text('Flutter REST API Example'),
        ),
        body: Center(
          child: FutureBuilder<Post>(
```

```
              future: post,
              builder: (context, abc) {
                if (abc.hasData) {
                  return Text(abc.data.title);
                } else if (abc.hasError) {
                  return Text("${abc.error}");
                }

                // By default, it show a loading spinner.
                return CircularProgressIndicator();
              },
            ),
          ),
        ),
      );
    }
  }

  Future<Post> fetchPost() async {
    final response = await http.get('Give your JSON file web link.');

    if (response.statusCode == 200) {
      // If the call to the server was successful (returns OK), parse
      return Post.fromJson(json.decode(response.body));
    } else {
      // If that call was not successful (response was unexpected), it
      throw Exception('Failed to load post');
    }
  }

  class Post {
    final int userId;
    final int id;
    final String title;
    final String description;

    Post({this.userId, this.id, this.title, this. description});

    factory Post.fromJson(Map<String, dynamic> json) {
      return Post(
        userId: json['userId'],
        id: json['id'],
        title: json['title'],
        description: json[' description'],
      );
```

```
    }
}
```

Json:

```
[
   {
      "userId": 01,
      "id": 1,
      "title": "iPhone",
      "description": "iPhone is the very stylist phone ever"
   },
   {
      "userId": 02,
      "id": 2,
      "title": "Pixel",
      "description": "Pixel is the most feature phone ever"
   },
   {
      "userId": 03,
      "id": 3,
      "title": "Laptop",
      "description": "Laptop is most popular development tool"
   },
   {
      "userId": 04,
      "id": 4,
      "title": "Tablet",
      "description": "Tablet is the most useful device used for
meeting"
   }
]
```

Cài ứng dụng cafedev để dễ dàng cập nhật tin và học lập trình mọi lúc mọi nơi tại đây.

Tài liệu từ cafedev:

- Full series tự học Flutter từ cơ bản tới nâng cao tại đây nha.
- **Các nguồn kiến thức MIỄN PHÍ VÔ GIÁ từ cafedev tại đây**

Nếu bạn thấy hay và hữu ích, bạn có thể tham gia các kênh sau của cafedev để nhận được nhiều hơn nữa:

- **Group Facebook**
- **Fanpage**
- **Youtube**
- **Instagram**
- **Twitter**
- **Linkedin**
- **Pinterest**
- **Trang chủ**

Chào thân ái và quyết thắng!

Đăng ký kênh youtube để ủng hộ Cafedev nha các bạn, Thanks you!

Bài trước

Tự học Flutter | Tìm hiểu về Google Maps trong Flutter

Bài tiếp theo

Tự học Flutter | Tìm hiểu về Database trong Flutter

David Xuân

https://cafedev.vn/

Tự học Flutter | Tìm hiểu về Google Maps trong Flutter

Flutter Google Maps

Bởi **David Xuân** - 3 Tháng Năm, 2021 692 0

Bản đồ được sử dụng để lấy thông tin về thế giới một cách đơn giản và trực quan. Nó giới thiệu các địa điểm trên thế giới bằng cách hiển thị hình dạng và kích thước, vị trí và khoảng cách giữa chúng. Chúng ta có thể thêm bản đồ vào ứng dụng của mình bằng cách sử dụng **plugin Google Maps Flutter** . Plugin này có thể tự động truy cập vào máy chủ Google Maps, hiển thị bản đồ và phản hồi các cử chỉ của người dùng. Nó cũng cho phép chúng ta thêm các điểm đánh dấu vào bản đồ của chúng ta.

Nội dung chính

1. Tại sao sử dụng Google Maps với Flutter?

Các nhà phát triển Flutter thích Google Maps cho ứng dụng của họ vì chúng **cung cấp hiệu suất gốc cho cả Android và iOS** . Nó cho phép chúng ta triển khai code một lần và cho phép chúng chạy code cho cả hai thiết bị (android và iOS). Plugin Google Maps Flutter được cung cấp trong widget Google Map hỗ trợ **InitialCameraPosition, maptype** và **onMapCreate** . Chúng ta có thể đặt vị trí của máy ảnh và điểm đánh dấu

ở bất kỳ nơi nào trên trái đất. Chúng ta có thể thiết kế điểm đánh dấu theo sự lựa chọn của chúng ta. Nó cũng đi kèm với một thuộc tính thu phóng trong một vị **trí phụ** để cung cấp khả năng phóng to trong chế độ xem bản đồ google trên trang đầu tiên.

Hãy để chúng ta xem từng bước cách thêm Google Maps vào ứng dụng Flutter.

Bước 1: Tạo một dự án mới. Mở dự án này trong IDE, điều hướng đến thư mục **lib** , sau đó mở tệp **pubspec.yaml** để thiết lập bản đồ.

Bước 2: Trong tệp pubspec.yaml, chúng ta cần thêm plugin Google Maps Flutter vào phần phụ thuộc, plugin này có sẵn dưới dạng **google_maps_flutter** trên pub.dartlang.org. Sau khi thêm phần phụ thuộc, hãy nhấp vào liên kết **lấy gói** để nhập thư viện trong **tệp main.dar** t.

```
dependencies:
  flutter:
    sdk: flutter
  cupertino_icons: ^0.1.2
  google_maps_flutter: ^0.5.21
```

Nó đảm bảo rằng chúng ta đã để lại hai khoảng trắng từ phía bên trái của phần phụ thuộc google_maps_flutter trong khi thêm các phần phụ thuộc.

Bước 3: Bước tiếp theo là lấy **khóa API cho dự án của bạn** . Nếu chúng ta đang sử dụng nền tảng Android, hãy làm theo hướng dẫn được cung cấp trên SDK Maps dành cho Android: Nhận Khóa API . Sau khi tạo khóa API, hãy thêm khóa đó vào tệp kê khai ứng dụng. Chúng ta có thể tìm thấy tệp này bằng cách điều hướng đến **android / app / src / main / AndroidManifest.xml** như sau:

```
<manifest ...
  <application ...
    <meta-data android:name="com.google.android.geo.API_KEY"
               android:value="YOUR ANDROID API KEY HERE"/>
```

Bước 4: Tiếp theo, nhập gói vào tệp Dart như bên dưới:

```
import 'package:google_maps_flutter/google_maps_flutter.dart';
```

Bước 5: Bây giờ, chúng ta đã sẵn sàng thêm widget GoogleMap để bắt đầu tạo giao diện người dùng để hiển thị bản đồ.

2. Thí dụ

Hãy để chúng ta hiểu nó với sự trợ giúp của một ví dụ.

```
import 'package:flutter/material.dart';
import 'package:google_maps_flutter/google_maps_flutter.dart';

void main() => runApp(MyApp());

class MyApp extends StatefulWidget {
  @override
  _MyAppState createState() => _MyAppState();
}

class _MyAppState extends State<MyApp> {
  GoogleMapController myController;

  final LatLng _center = const LatLng(45.521563, -122.677433);

  void _onMapCreated(GoogleMapController controller) {
    myController = controller;
  }

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Flutter Maps Demo'),
          backgroundColor: Colors.green,
        ),
        body: Stack(
          children: <Widget>[
            GoogleMap(
              onMapCreated: _onMapCreated,
              initialCameraPosition: CameraPosition(
                target: _center,
```

```
                zoom: 10.0,
              ),
            ),
            Padding(
              padding: const EdgeInsets.all(14.0),
              child: Align(
                alignment: Alignment.topRight,
                child: FloatingActionButton(
                  onPressed: () => print('You have pressed the butto
                  materialTapTargetSize: MaterialTapTargetSize.padde
                  backgroundColor: Colors.green,
                  child: const Icon(Icons.map, size: 30.0),
                ),
              ),
            ),
          ],
        ),
      ),
    );
  }
}
```

Trong đoạn code trên, chúng ta đã nhận thấy các điều khoản sau:

mapController : Nó tương tự như các bộ điều khiển khác mà chúng ta đã thấy trong Flutter. Nó kiểm soát tất cả các hoạt động trên lớp GoogleMap. Ở đây, nó quản lý chức năng của máy ảnh, chẳng hạn như vị trí, hoạt ảnh, phóng to, v.v.

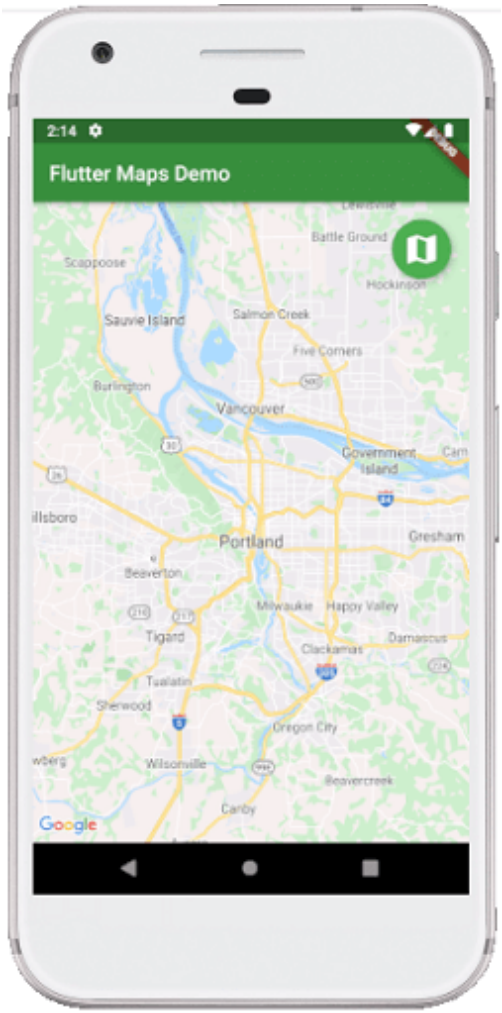
onMapCreate : Nó là một phương thức được gọi để tạo một bản đồ và lấy MapController làm đối số.

InitialCameraPosition : Đây là một tham số bắt buộc đặt vị trí camera từ nơi chúng ta muốn bắt đầu. Nó cho phép chúng ta thiết lập phần nào của thế giới mà chúng ta muốn chỉ trên bản đồ.

Stack: Nó được sử dụng để đặt các widget Flutter khác lên trên widget bản đồ.

Đầu ra:

Khi chúng ta chạy ứng dụng, nó sẽ trả về giao diện người dùng của màn hình như ảnh chụp màn hình bên dưới:



3. Làm thế nào để thay đổi diện mạo bản đồ?

Chúng ta có thể thay đổi giao diện bản đồ như chế độ xem thông thường, chế độ xem vệ tinh, v.v. bằng cách sử dụng **thuộc** tính **mapType**. Thuộc tính này cho phép các nhà phát triển hiển thị loại ô bản đồ. widget GoogleMap chủ yếu cung cấp năm loại ô, được đưa ra bên dưới:

- **none** : Nó không hiển thị bất kỳ ô bản đồ nào.
- **normal** : Nó hiển thị các ô trên bản đồ với giao thông, nhãn và thông tin địa hình tinh tế.
- **satellite**: Nó hiển thị hình ảnh vệ tinh (ảnh hàng không) của vị trí.
- **terrain**: Nó hiển thị các tính năng vật lý cụ thể của một khu vực đất (cho biết loại và độ cao của địa hình).
- **hybrid** : Nó hiển thị các hình ảnh vệ tinh với một số nhãn hoặc lớp phủ.

Chúng ta có thể thực hiện việc này bằng cách tạo một biến _ **currentMapType** trong đoạn code trên để hiển thị loại bản đồ hiện tại và sau đó thêm **mapType:**

_currentMapType vào widget **GoogleMap** .

```
MapType _currentMapType = MapType.normal;

@override
Widget build(BuildContext context) {
  return MaterialApp(
    GoogleMap(
      mapType: _currentMapType,
    ),
  );
}
```

Tiếp theo, chúng ta phải thêm một phương thức để sửa đổi giá trị của _currentMapType bên trong một lời gọi hàm **setState**(). Phương thức này sẽ cập nhật giao diện bản đồ để khớp với giá trị mới của biến _currentMapType.

```
1  void _onMapTypeButtonPressed() {
2    setState(() {
3      _currentMapType = _currentMapType = = MapType.normal
4          ? MapType.satellite
5          : MapType.normal;
6    });
7  }
```

Cuối cùng, thay thế thuộc tính **onPressed** bằng _ **onMapTypeButtonPressed** .

```
1  child: FloatingActionButton(
2    onPressed: _onMapTypeButtonPressed
3  ),
```

Hãy cho chúng ta xem code hoàn chỉnh để thay đổi giao diện bản đồ. Mở tệp Dart và thay thế bằng code dưới đây:

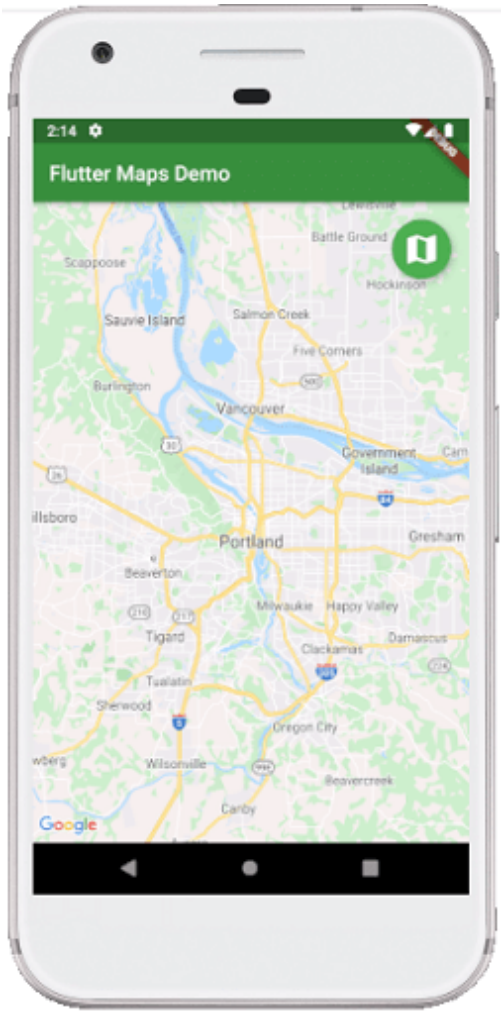
```
    import 'package:flutter/material.dart';
    import 'package:google_maps_flutter/google_maps_flutter.dart';

    void main() => runApp(MyApp());

    class MyApp extends StatefulWidget {
```

```
  @override
  _MyAppState createState() => _MyAppState();
}

class _MyAppState extends State<MyApp> {
  GoogleMapController myController;

  final LatLng _center = const LatLng(45.521563, -122.677433);

  MapType _currentMapType = MapType.normal;

  void _onMapTypeButtonPressed() {
    setState(() {
      _currentMapType = _currentMapType == MapType.normal
          ? MapType.satellite
          : MapType.normal;
    });
  }

  void _onMapCreated(GoogleMapController controller) {
    myController = controller;
  }

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Flutter Maps Demo'),
          backgroundColor: Colors.green,
        ),
        body: Stack(
          children: <Widget>[
            GoogleMap(
              onMapCreated: _onMapCreated,
              initialCameraPosition: CameraPosition(
                target: _center,
                zoom: 10.0,
              ),
              mapType: _currentMapType
            ),
            Padding(
              padding: const EdgeInsets.all(14.0),
              child: Align(
                alignment: Alignment.topRight,
```

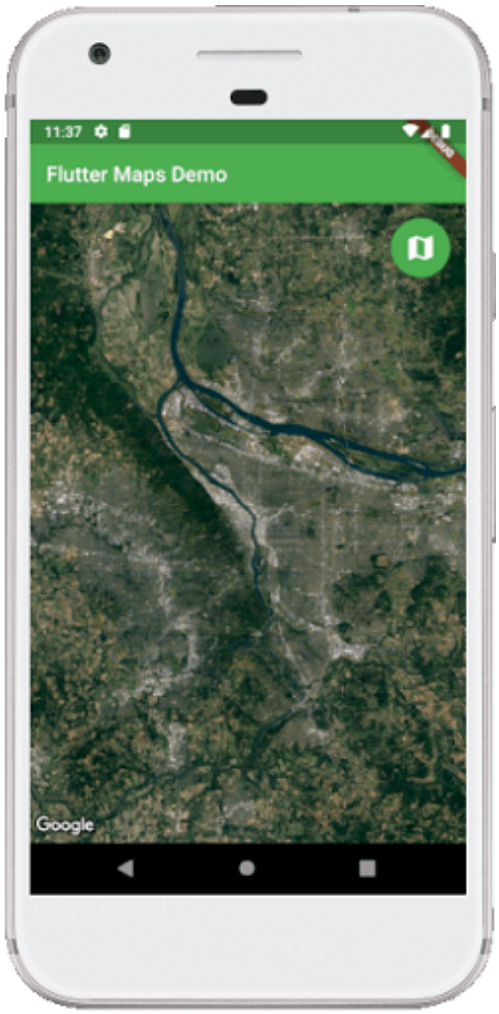
```
                child: FloatingActionButton(
                  onPressed: _onMapTypeButtonPressed,
                  materialTapTargetSize: MaterialTapTargetSize.padde
                  backgroundColor: Colors.green,
                  child: const Icon(Icons.map, size: 30.0),
                ),
              ),
            ),
          ],
        ),
      ),
    );
  }
}
```

Đầu ra:

Khi chúng ta chạy ứng dụng, nó sẽ trả về giao diện người dùng của màn hình như ảnh chụp màn hình bên dưới:



Nếu chúng ta nhấp vào biểu tượng bản đồ, chúng ta sẽ nhận được hình ảnh vệ tinh của vị trí đã chỉ định của chúng ta. Xem hình ảnh bên dưới:



4. Làm thế nào để thêm một điểm đánh dấu trên bản đồ?

Điểm đánh dấu xác định vị trí trên bản đồ. Chúng ta có thể thêm điểm đánh dấu trên bản đồ bằng cách sử dụng thuộc tính **điểm đánh dấu** bên trong widget GoogleMap. Nói chung, điểm đánh dấu được hiển thị khi chúng ta di chuyển máy ảnh của mình từ vị trí này sang vị trí khác. **Ví dụ:** nếu chúng ta muốn chuyển từ Ba Lan đến California, chúng ta có thể thấy điểm đánh dấu trên vị trí cụ thể đó khi máy ảnh chuyển đến California.

Chúng ta có thể thêm điểm đánh dấu bằng cách tạo biến _ **điểm đánh dấu** sẽ lưu trữ điểm đánh dấu của bản đồ và sau đó đặt biến này làm thuộc tính điểm đánh dấu trong widget con GoogleMap.

```
final Set<Marker> _markers = {};
@override
Widget build(BuildContext context) {
```

```
  return MaterialApp(
    GoogleMap(
      markers: _markers,
    ),
  );
}
```

Tiếp theo, cần theo dõi vị trí camera hiện tại trên bản đồ bằng cách thêm code dưới đây:

```
LatLng _currentMapPosition = _center;

void _onCameraMove(CameraPosition position) {
  _currentMapPosition = position.target;
}

@override
Widget build(BuildContext context) {
  return MaterialApp(
    GoogleMap(
      onCameraMove: _onCameraMove,
    ),
  );
}
```

Cuối cùng, chúng ta cần thêm một điểm đánh dấu trong bản đồ bằng cách sửa đổi các dấu đánh dấu bên trong một **lệnh** gọi hàm **setState**().

```
void _onAddMarkerButtonPressed() {
  setState(() {
    _markers.add(Marker(
      markerId: MarkerId(_lastMapPosition.toString()),
      position: _currentMapPosition,
      infoWindow: InfoWindow(
        title: 'Nice Place'
      ),
      icon: BitmapDescriptor. defaultMarkerWithHue(BitmapDescriptor.
    ));
  });
}
```

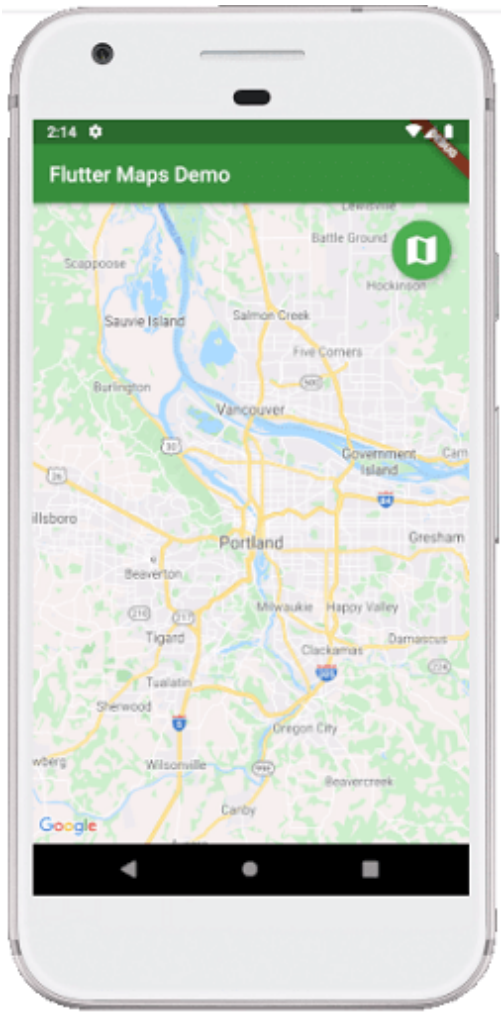
Hãy cho chúng ta xem code hoàn chỉnh để thêm điểm đánh dấu vào bản đồ. Mở tệp Dart và thay thế bằng code dưới đây:

```
import 'package:flutter/material.dart';
import 'package:google_maps_flutter/google_maps_flutter.dart';

void main() => runApp(MyApp());

class MyApp extends StatefulWidget {
  @override
  _MyAppState createState() => _MyAppState();
}

class _MyAppState extends State<MyApp> {
  GoogleMapController mapController;
  static final LatLng _center = const LatLng(45.521563, -122.677433)
  final Set<Marker> _markers = {};
  LatLng _currentMapPosition = _center;

  void _onAddMarkerButtonPressed() {
    setState(() {
      _markers.add(Marker(
        markerId: MarkerId(_currentMapPosition.toString()),
        position: _currentMapPosition,
        infoWindow: InfoWindow(
          title: 'Nice Place',
          snippet: 'Welcome to Poland'
        ),
        icon: BitmapDescriptor.defaultMarker,
      ));
    });
  }

  void _onCameraMove(CameraPosition position) {
    _currentMapPosition = position.target;
  }

  void _onMapCreated(GoogleMapController controller) {
    mapController = controller;
  }

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
```

```
        home: Scaffold(
          appBar: AppBar(
            title: Text('Flutter Maps Demo'),
            backgroundColor: Colors.green,
          ),
          body: Stack(
            children: <Widget>[
              GoogleMap(
                onMapCreated: _onMapCreated,
                initialCameraPosition: CameraPosition(
                  target: _center,
                  zoom: 10.0,
                ),
                markers: _markers,
                onCameraMove: _onCameraMove
              ),
              Padding(
                padding: const EdgeInsets.all(14.0),
                child: Align(
                  alignment: Alignment.topRight,
                  child: FloatingActionButton(
                    onPressed: _onAddMarkerButtonPressed,
                    materialTapTargetSize: MaterialTapTargetSize.padde
                    backgroundColor: Colors.green,
                    child: const Icon(Icons.map, size: 30.0),
                  ),
                ),
              ),
            ],
          ),
        ),
      );
    }
  }
```

Đầu ra:

Chạy ứng dụng và chúng ta sẽ nhận được giao diện người dùng của màn hình làm ảnh chụp màn hình đầu tiên. Khi chúng ta nhấp vào nút, nó sẽ hiển thị điểm đánh dấu ở vị trí cuối cùng trên bản đồ. Khi chúng ta chạm vào điểm đánh dấu, thông tin được hiển thị cung cấp tiêu đề và đoạn trích về vị trí.



Cài ứng dụng cafedev để dễ dàng cập nhật tin và học lập trình mọi lúc mọi nơi tại đây.

Tài liệu từ cafedev:

- Full series tự học Flutter từ cơ bản tới nâng cao tại đây nha.
- **Các nguồn kiến thức MIỄN PHÍ VÔ GIÁ từ cafedev tại đây**

Nếu bạn thấy hay và hữu ích, bạn có thể tham gia các kênh sau của cafedev để nhận được nhiều hơn nữa:

- **Group Facebook**
- **Fanpage**
- **Youtube**
- **Instagram**
- **Twitter**

- **Linkedin**
- **Pinterest**
- **Trang chủ**

Chào thân ái và quyết thắng!

Đăng ký kênh youtube để ủng hộ Cafedev nha các bạn, Thanks you!

Bài trước

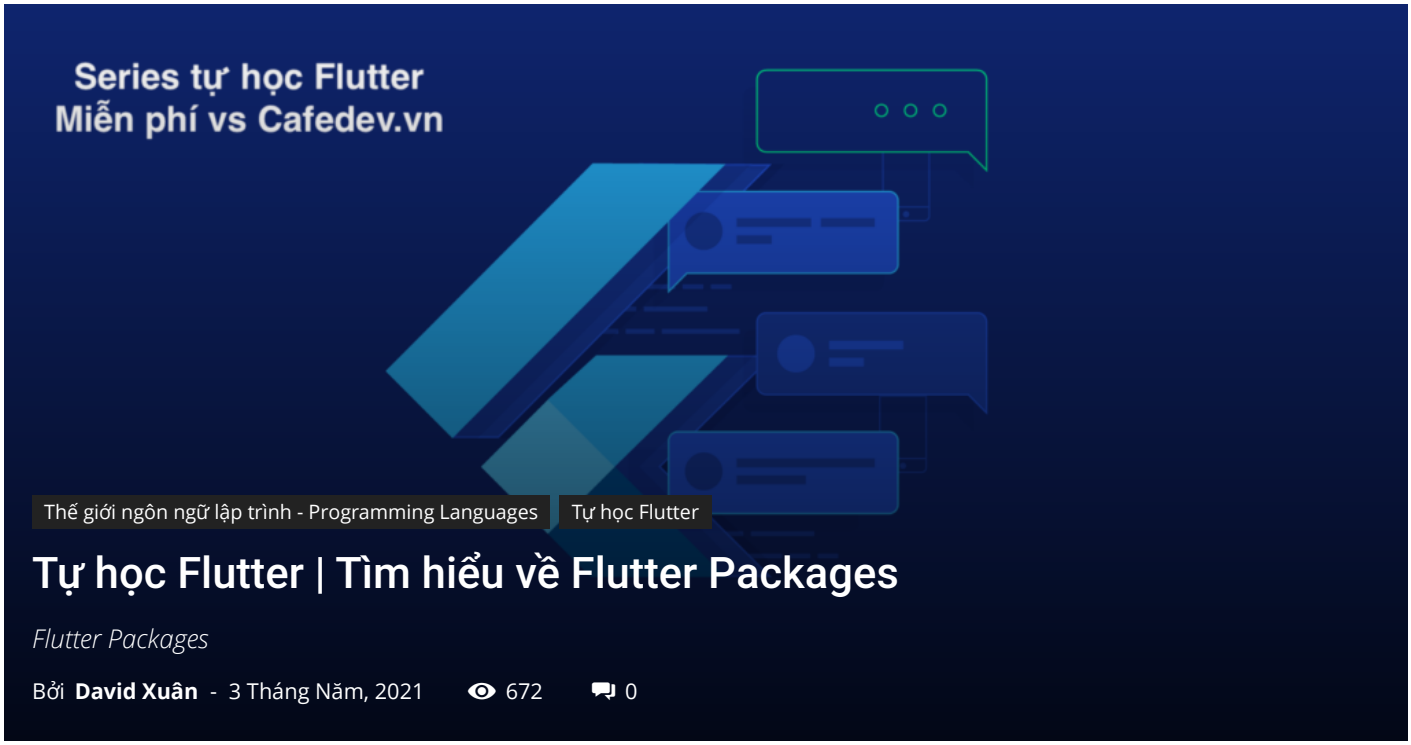
Tự học Flutter | Tìm hiểu về Flutter Packages

Bài tiếp theo

Tự học Flutter | Tìm hiểu về API REST trong Flutter

David Xuân

https://cafedev.vn/



Tự học Flutter | Tìm hiểu về Flutter Packages

Flutter Packages

Bởi **David Xuân** - 3 Tháng Năm, 2021 672 0

Gói(Packages) là một không gian tên chứa một nhóm các loại lớp, giao diện và gói con tương tự nhau. Chúng ta có thể coi các gói tương tự như các thư mục khác nhau trên máy tính của mình, nơi chúng ta có thể giữ phim trong một thư mục, hình ảnh trong thư mục khác, phần mềm trong thư mục khác, v.v. Trong Flutter, Dart tổ chức và chia sẻ một bộ chức năng thông qua một gói. Flutter luôn hỗ trợ các gói chia sẻ, được đóng góp bởi các nhà phát triển khác cho hệ sinh thái Flutter và Dart. Các gói cho phép chúng ta xây dựng ứng dụng mà không cần phải phát triển mọi thứ từ đầu.

Cấu trúc chung của gói được đưa ra bên dưới (Giả sử gói demo là mycustom_package):

lib / src / *: Nó chứa các tệp code Dart riêng.

lib / mydemo_package.dart: Đây là một tệp code Dart chính. Chúng ta có thể nhập nó vào một ứng dụng như sau:

```
import 'package:mycustom_package/mycustom_package.dart'
```

Chúng ta cũng có thể xuất bất kỳ tệp code nào khác vào tệp code chính theo cú pháp dưới đây:

```
Export src/my_code.dart
```

lib / *: Đây là một thư mục chứa code công khai trong gói. Chúng ta có thể truy cập code này như sau:

```
import 'package:mydemo_package/sub_folder/custom_file.dart'
```

pubspec.yaml: Đó là tệp cấu hình của dự án sẽ sử dụng nhiều trong quá trình làm việc với dự án Flutter. Tệp này chứa:

- Cài đặt chung của dự án như tên, mô tả và phiên bản của dự án.
- Sự phụ thuộc của dự án.
- Tài sản dự án (ví dụ: hình ảnh).

1. Các loại gói(Packages)

Theo chức năng, chúng ta có thể phân loại gói thành hai loại:

1. Gói Dart
2. Gói plugin

Gói Dart: Là một gói chung, được viết bằng ngôn ngữ Dart, chẳng hạn như gói đường dẫn. Gói này có thể được sử dụng trong cả môi trường, đó là nền tảng web hoặc di động. Nó cũng chứa một số chức năng cụ thể của Flutter và do đó có sự phụ thuộc vào khuôn khổ Flutter, chẳng hạn như **gói fluro** .

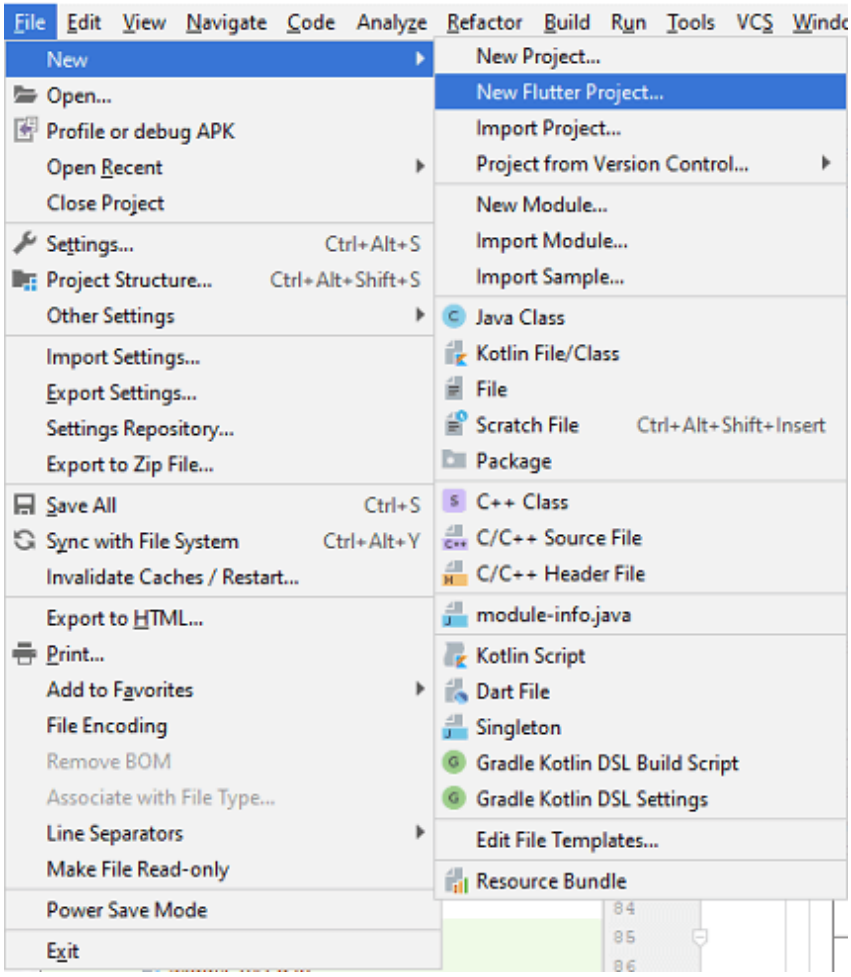
Gói plugin: Là một gói Dart chuyên biệt bao gồm một API được viết bằng code Dart và phụ thuộc vào framework Flutter. Nó có thể được kết hợp với triển khai nền tảng cụ

thể cho một nền tảng cơ bản như Android (sử dụng Java hoặc Kotlin) và iOS (sử dụng Objective C hoặc Swift). Ví dụ về gói này là gói plugin bộ chọn hình ảnh và pin.

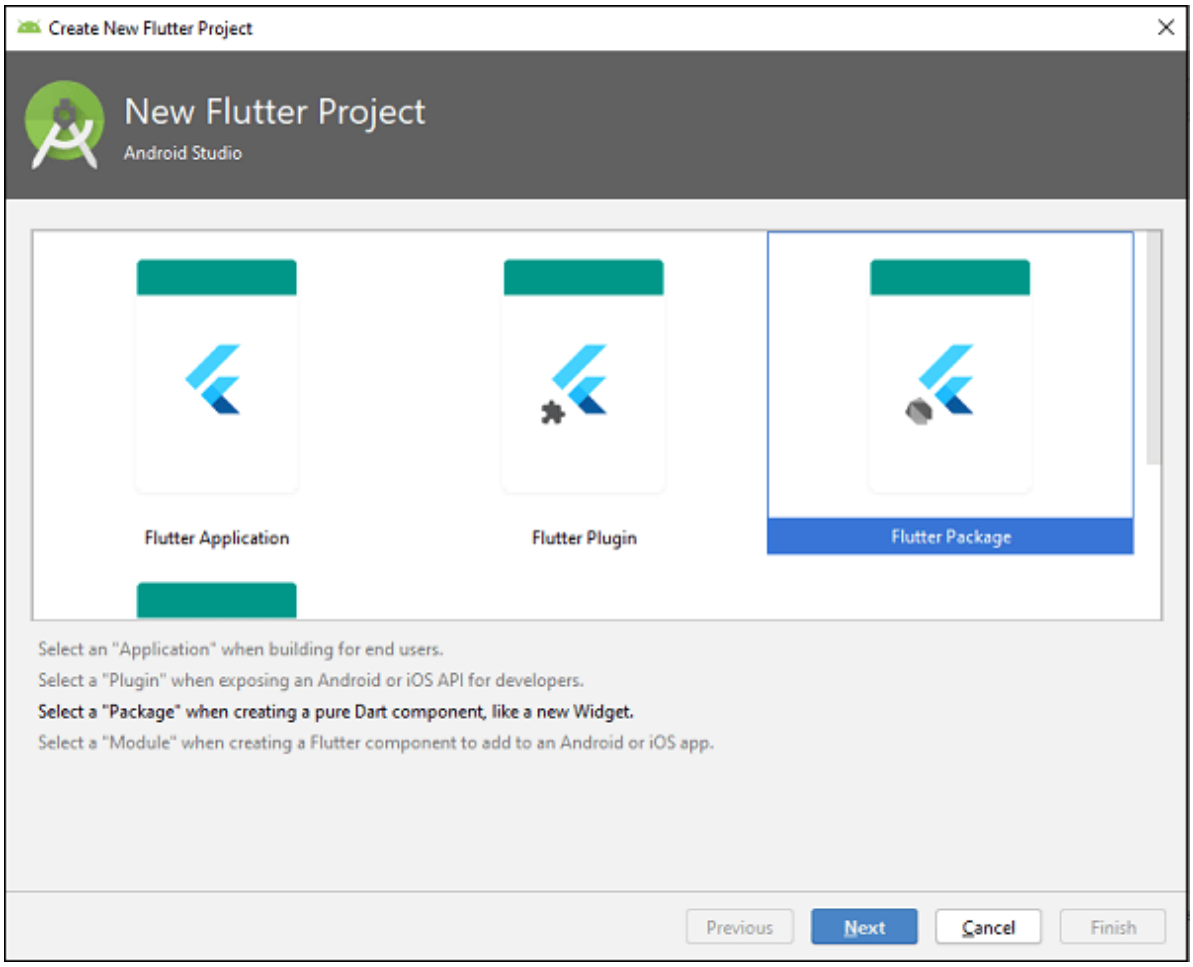
2. Phát triển gói hoặc plugin Flutter

Việc phát triển một plugin hoặc gói Flutter tương tự như tạo một ứng dụng Dart hoặc gói Dart. Tuy nhiên, nó có một số ngoại lệ có nghĩa là plugin luôn sử dụng API hệ thống cụ thể cho một nền tảng như Android hoặc iOS để có được chức năng cần thiết. Bây giờ, chúng ta hãy xem từng bước cách phát triển một gói trong Flutter.

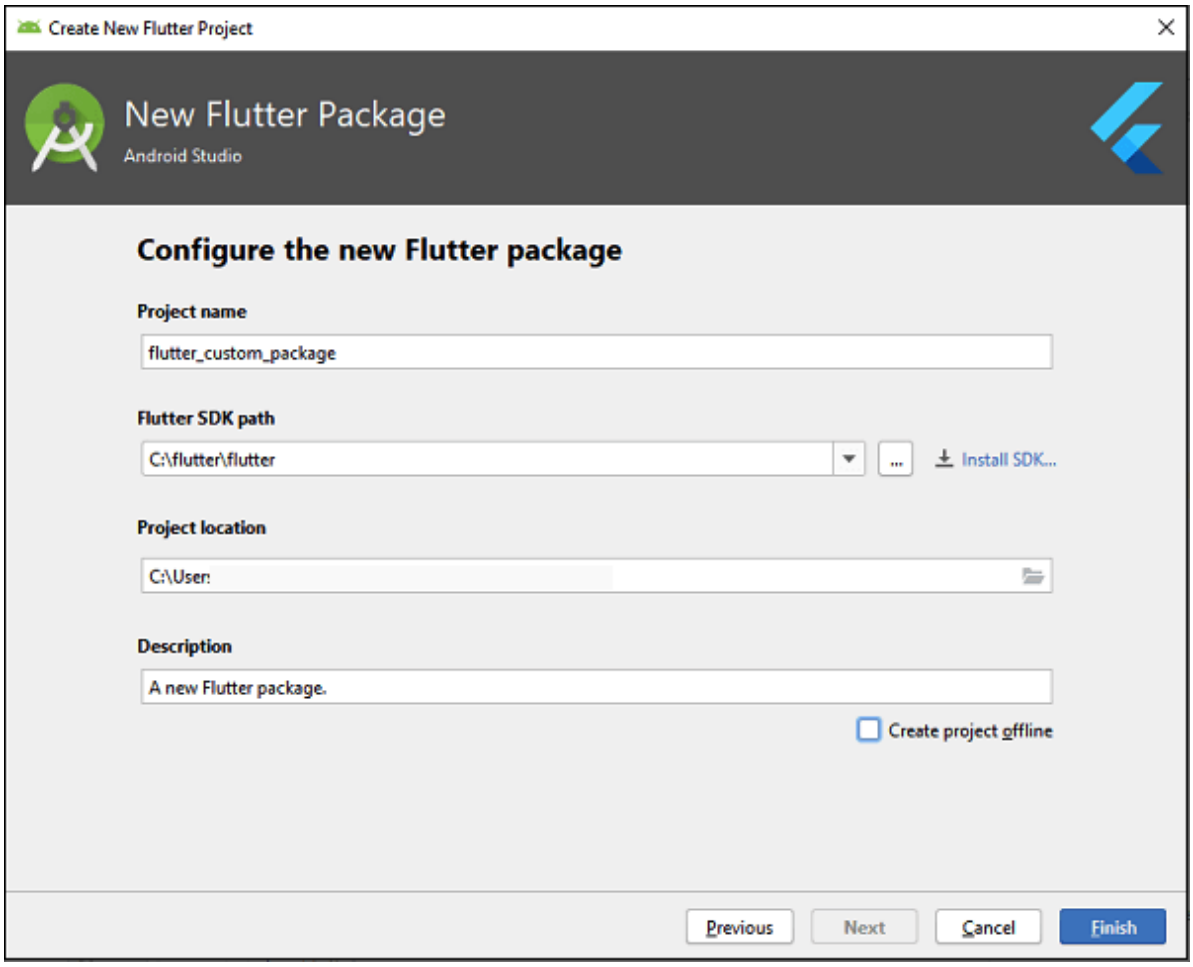
Bước 1: Đầu tiên, mở Android Studio và nhấp vào menu Tệp -> Chọn một dự án Flutter mới. Một hộp thoại sẽ xuất hiện trên màn hình của bạn.



Bước 2: Trong hộp thoại này, bạn cần chọn một tùy chọn Dự án Flutter Mới, như thể hiện trong hình dưới đây và nhấp vào **Tiếp theo** .



Bước 3: Trong hộp thoại tiếp theo, nhập tất cả các chi tiết về gói của bạn, chẳng hạn như tên dự án, vị trí của dự án và mô tả dự án. Sau khi điền tất cả các chi tiết, hãy nhấp vào Hoàn tất.



Bước 4: Cuối cùng, dự án của bạn đã được tạo. Bây giờ, mở tệp flay_custom_package.dart và xóa code mặc định được tạo tại thời điểm tạo dự án. Sau đó, chèn code sau. Đoạn code này tạo một gói **hộp cảnh báo** .

```
library flutter_custom_package;

import 'package:flutter/material.dart';

class CustomPackageAlertBox {
  static Future showCustomAlertBox({
    @required BuildContext context,
    @required Widget willDisplayWidget,
  }) {
    assert(context != null, "If context is null!!");
    assert(willDisplayWidget != null, "If willDisplayWidget is null!
    return showDialog(
        context: context,
        builder: (context) {
          return AlertDialog(
            shape: RoundedRectangleBorder(
              borderRadius: BorderRadius.all(Radius.circular(20)),
```

```
            ),
            content: Column(
              mainAxisSize: MainAxisSize.min,
              children: <Widget>[
                willDisplayWidget,
                MaterialButton(
                  color: Colors.white70,
                  child: Text('Close Alert'),
                  onPressed: () {
                    Navigator.of(context).pop();
                  },
                )
              ],
            ),
            elevation: 12,
          );
        });
  }
}
```

Bây giờ, bạn cần kiểm tra gói mới tạo của mình. Để kiểm tra gói, hãy tạo một dự án mới. Trong dự án này, trước tiên, hãy mở tệp **pubspec.yaml** và đoạn code sau trong phần phụ thuộc.

```
dependencies:
  flutter:
    sdk: flutter
  flutter_custom_package:
    path: ../
```

Khi bạn thêm gói tùy chỉnh trong tệp pubspec.yaml, Android Studio sẽ cảnh báo bạn cập nhật tệp này. Để cập nhật tệp, hãy nhấp vào Nhận phần phụ thuộc và đảm bảo rằng bạn có kết nối internet trong quá trình cập nhật tệp. Android Studio tự động tải gói từ internet và định cấu hình gói cho ứng dụng của bạn. Bây giờ, bạn có thể sử dụng gói

```
import 'package: flutter_custom_package/flutter_custom_package.dart';
```

3. Làm thế nào để xuất bản một gói

Khi bạn đã triển khai thành công một gói, bạn có thể xuất bản gói đó trên pub.dev , để bất kỳ ai cũng có thể sử dụng dễ dàng trong dự án.

Trước khi xuất bản gói, hãy đảm bảo rằng nội dung của tệp pubspec.yaml, README.md và CHANGELOG.md là hoàn chỉnh và chính xác.

Tiếp theo, chạy lệnh sau trong cửa sổ đầu cuối để phân tích mọi giai đoạn của gói.

```
$ flutter pub publish --dry-run
```

Cuối cùng, bạn cần chạy lệnh sau để **xuất bản** gói.

```
$ flutter pub publish
```

Cài ứng dụng cafedev để dễ dàng cập nhật tin và học lập trình mọi lúc mọi nơi tại đây.

Tài liệu từ cafedev:

- Full series tự học Flutter từ cơ bản tới nâng cao tại đây nha.
- **Các nguồn kiến thức MIỄN PHÍ VÔ GIÁ từ cafedev tại đây**

Nếu bạn thấy hay và hữu ích, bạn có thể tham gia các kênh sau của cafedev để nhận được nhiều hơn nữa:

- **Group Facebook**
- **Fanpage**
- **Youtube**
- **Instagram**
- **Twitter**
- **Linkedin**
- **Pinterest**
- **Trang chủ**

Chào thân ái và quyết thắng!

Đăng ký kênh youtube để ủng hộ Cafedev nha các bạn, Thanks you!

Bài trước

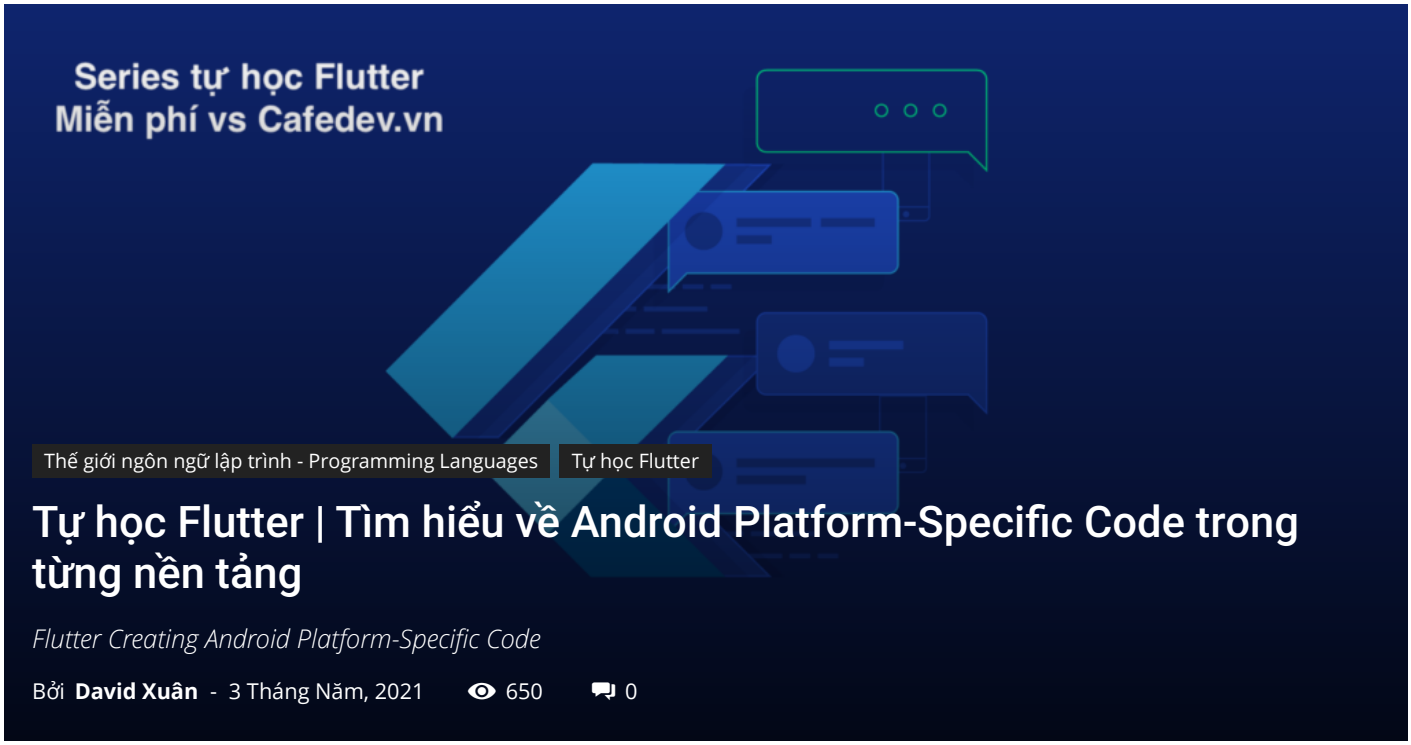
Tự học Flutter | Tìm hiểu về Android Platform-Specific Code trong từng nền tảng

Bài tiếp theo

Tự học Flutter | Tìm hiểu về Google Maps trong Flutter

David Xuân

https://cafedev.vn/



Thế giới ngôn ngữ lập trình - Programming Languages | Tự học Flutter

Tự học Flutter | Tìm hiểu về Android Platform-Specific Code trong từng nền tảng

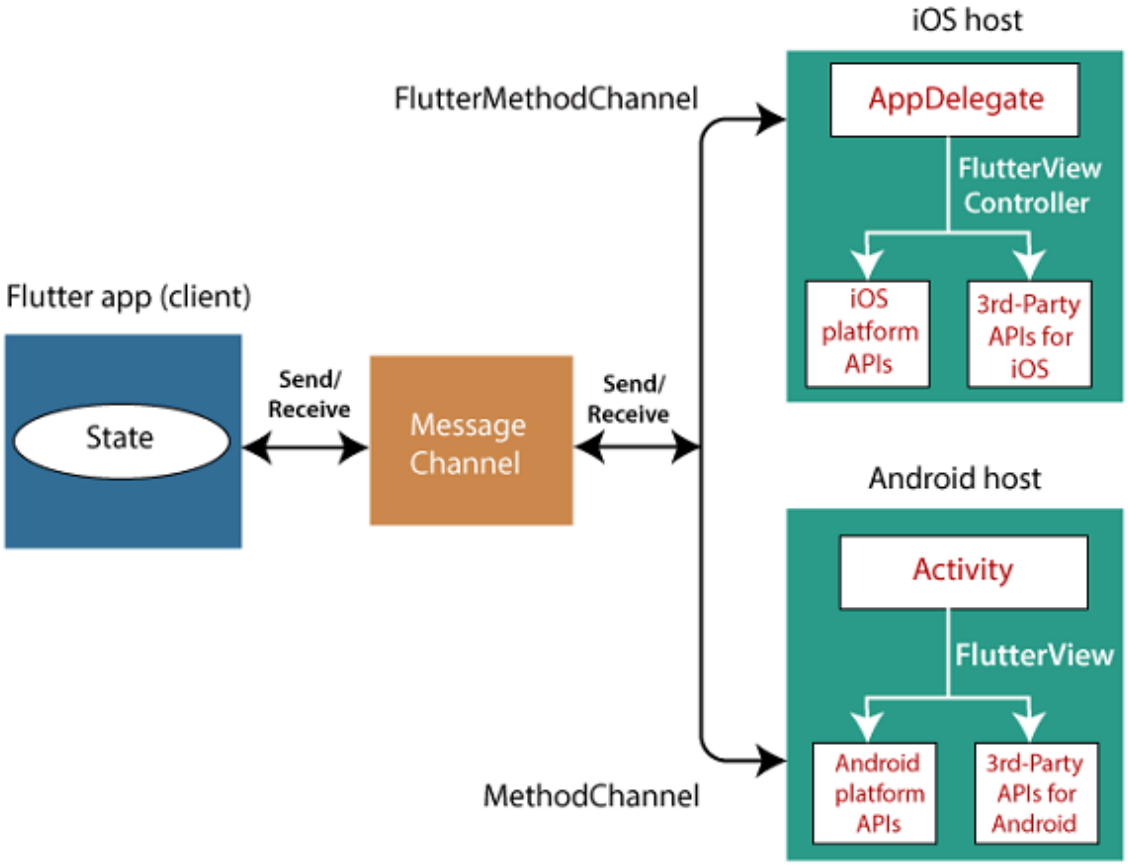
Flutter Creating Android Platform-Specific Code

Bởi **David Xuân** - 3 Tháng Năm, 2021 650 0

Trong phần này, chúng ta sẽ xem cách chúng ta có thể viết code tùy chỉnh theo nền tảng cụ thể trong Flutter. Flutter là một framework công tác tuyệt vời, cung cấp cơ chế xử lý / truy cập các tính năng dành riêng cho nền tảng. Tính năng này cho phép nhà phát triển mở rộng chức năng của framework Flutter. Một số chức năng thiết yếu dành riêng cho nền tảng có thể được truy cập dễ dàng thông qua framework là máy ảnh, mức pin, trình duyệt, v.v.

Flutter sử dụng một hệ thống linh hoạt để gọi API dành riêng cho nền tảng có sẵn trên Android bằng code Java hoặc Kotlin hoặc trong code Objective-C hoặc Swift trên iOS. Ý tưởng chung về việc truy cập code dành riêng cho nền tảng trong Flutter là thông qua **giao thức nhắn tin(messaging protocol).** Thông báo được chuyển giữa **máy khách** (Giao diện người dùng) và **Máy chủ** (Nền tảng) bằng cách sử dụng kênh thông báo chung. Nó có nghĩa là các khách hàng gửi một tin nhắn đến Host bằng cách sử dụng kênh tin nhắn này. Tiếp theo, Host lắng nghe trên kênh thông báo đó, nhận thông báo, thực hiện các chức năng phù hợp và cuối cùng trả kết quả cho máy khách.

Sơ đồ khối sau đây cho thấy kiến trúc code nền tảng cụ thể phù hợp.



Kênh nhắn tin sử dụng **codec** tin nhắn tiêu chuẩn (lớp StandardMessageCodec), hỗ trợ tuần tự hóa nhị phân hiệu quả của các giá trị như JSON, chẳng hạn như chuỗi, Boolean, số, bộ đệm byte, Danh sách và Bản đồ, v.v. Việc tuần tự hóa và giải code hóa các giá trị này hoạt động tự động giữa máy khách và máy chủ khi bạn gửi và nhận giá trị.

Bảng sau đây cho biết cách nhận các giá trị Dart trên nền tảng Android và iOS và ngược lại:

Dart	**Android**	**iOS**
null	null	nil (NSNull khi được lồng vào nhau)
bool	java.lang.Boolean	NSNumber NSNumberWithBool:
int	java.lang.Integer	NSNumber NSNumberWithInt:
double	java.lang.Double	NSNumber NSNumberWithDouble:
String	java.lang.String	NSString:
Uint8List	byte []	FlutterStandardTypedData typedDataWithBytes:
Int32List	int []	FlutterStandardTypedData typedDataWithInt32:
Int64List	long[]	FlutterStandardTypedData typedDataWithInt64:
Float64List	double[]	FlutterStandardTypedData typedDataWithFloat64:

List	java.util.ArrayList	NSArray
Map	java.util.HashMAp	NSDictionary

Hãy để chúng ta tạo một ứng dụng đơn giản để trình bày cách gọi một API dành riêng cho nền tảng để mở trình duyệt. Để thực hiện việc này, chúng ta cần tạo một dự án Flutter trong Android Studio và chèn code sau vào tệp **main.dart** .

```
import 'package:flutter/material.dart';
import 'dart:async';
import 'package:flutter/services.dart';

void main() => runApp(MyApp());
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter DemoApplication',
      theme: ThemeData(
        primarySwatch: Colors.green,
      ),
      home: MyHomePage(
          title: 'Flutter Platform Specific Page'
      ),
    );
  }
}
class MyHomePage extends StatelessWidget {
  MyHomePage({Key key, this.title}) : super(key: key);
  final String title;
  static const platform = const MethodChannel('flutterplugins.javatp
  Future<void> _openBrowser() async {
    try {
      final int result = await platform.invokeMethod('openBrowser',
        'url': "https://www.cafedev.vn"
      });
    }
    on PlatformException catch (e) {
      // Unable to open the browser
      print(e);
    }
  }
  @override
  Widget build(BuildContext context) {
```

```
      return Scaffold(
        appBar: AppBar(
          title: Text(this.title),
        ),
        body: Center(
          child: RaisedButton(
            child: Text('Click Here'),
            onPressed: _openBrowser,
          ),
        ),
      );
    }
  }
```

Trong tệp ở trên, chúng ta đã nhập tệp **service.dart** , tệp này bao gồm chức năng gọi code dành riêng cho nền tảng. Trong tiện ích **MyHomePage** , chúng ta đã tạo một kênh tin nhắn và viết một phương thức _ **openBrowser** để gọi code dành riêng cho nền tảng.

```
Future<void> _openBrowser() async {
  try {
    final int result = await platform.invokeMethod('openBrowser',
<String, String>{
      'url': "https://www.cafedev.vn"
    });
  }
  on PlatformException catch (e) {
    // Unable to open the browser
 print(e);
  }
}
```

Cuối cùng, chúng ta đã tạo một nút để mở trình duyệt.

Bây giờ, chúng ta cần cung cấp triển khai nền tảng tùy chỉnh cụ thể. Để thực hiện việc này, hãy điều hướng đến **thư mục** Android của dự án Flutter của bạn và chọn tệp Java hoặc Kotlin và chèn code sau vào tệp **MainActivity** . code này có thể thay đổi theo ngôn ngữ Java hoặc Kotlin.

```
package com.javatpoint.flutterplugins.flutter_demoapplication

import android.app.Activity
import android.content.Intent
import android.net.Uri
import android.os.Bundle
import io.flutter.app.FlutterActivity
import io.flutter.plugin.common.MethodCall
import io.flutter.plugin.common.MethodChannel
import io.flutter.plugin.common.MethodChannel.MethodCallHandler
import io.flutter.plugin.common.MethodChannel.Result
import io.flutter.plugins.GeneratedPluginRegistrant

 class MainActivity:FlutterActivity() {
    override fun onCreate(savedInstanceState:Bundle?) {
        super.onCreate(savedInstanceState)
        GeneratedPluginRegistrant.registerWith(this)
        MethodChannel(flutterView, CHANNEL).setMethodCallHandler { c
        val url = call.argument<String>("url")
        if (call.method == "openBrowser") {
            openBrowser(call, result, url)
        } else {
            result.notImplemented()
        }
    }
 }
 private fun openBrowser(call:MethodCall, result:Result, url:String?
    val activity = this
    if (activity == null)
    {
        result.error("UNAVAILABLE", "It cannot open the browser with
        return
    }
    val intent = Intent(Intent.ACTION_VIEW)
    intent.data = Uri.parse(url)
    activity!!.startActivity(intent)
    result.success(true as Any)
 }

 companion object {
    private val CHANNEL = "flutterplugins.javatpoint.com/browser"
 }
}
```

Trong tệp **MainActivity.kt** , chúng ta đã tạo một phương thức **openBrowser()** để mở trình duyệt.

```
private fun openBrowser(call:MethodCall, result:Result, url:String?)
    val activity = this
    if (activity == null)
    {
        result.error("UNAVAILABLE", "It cannot open the browser with
        return
    }
    val intent = Intent(Intent.ACTION_VIEW)
    intent.data = Uri.parse(url)
    activity!!.startActivity(intent)
    result.success(true as Any)
 }
```

Đầu ra

Bây giờ, chạy ứng dụng trong studio android của bạn, bạn sẽ nhận được kết quả sau. Khi bạn nhấp vào nút **Nhấp vào đây(Click here),** bạn có thể thấy rằng màn hình trang chủ của trình duyệt được khởi chạy.

Cài ứng dụng cafedev để dễ dàng cập nhật tin và học lập trình mọi lúc mọi nơi tại đây.

Tài liệu từ cafedev:

- Full series tự học Flutter từ cơ bản tới nâng cao tại đây nha.
- **Các nguồn kiến thức MIỄN PHÍ VÔ GIÁ từ cafedev tại đây**

Nếu bạn thấy hay và hữu ích, bạn có thể tham gia các kênh sau của cafedev để nhận được nhiều hơn nữa:

- **Group Facebook**
- **Fanpage**
- **Youtube**
- **Instagram**
- **Twitter**
- **Linkedin**

- **Pinterest**
- **Trang chủ**

Chào thân ái và quyết thắng!

Đăng ký kênh youtube để ủng hộ Cafedev nha các bạn, Thanks you!

Bài trước

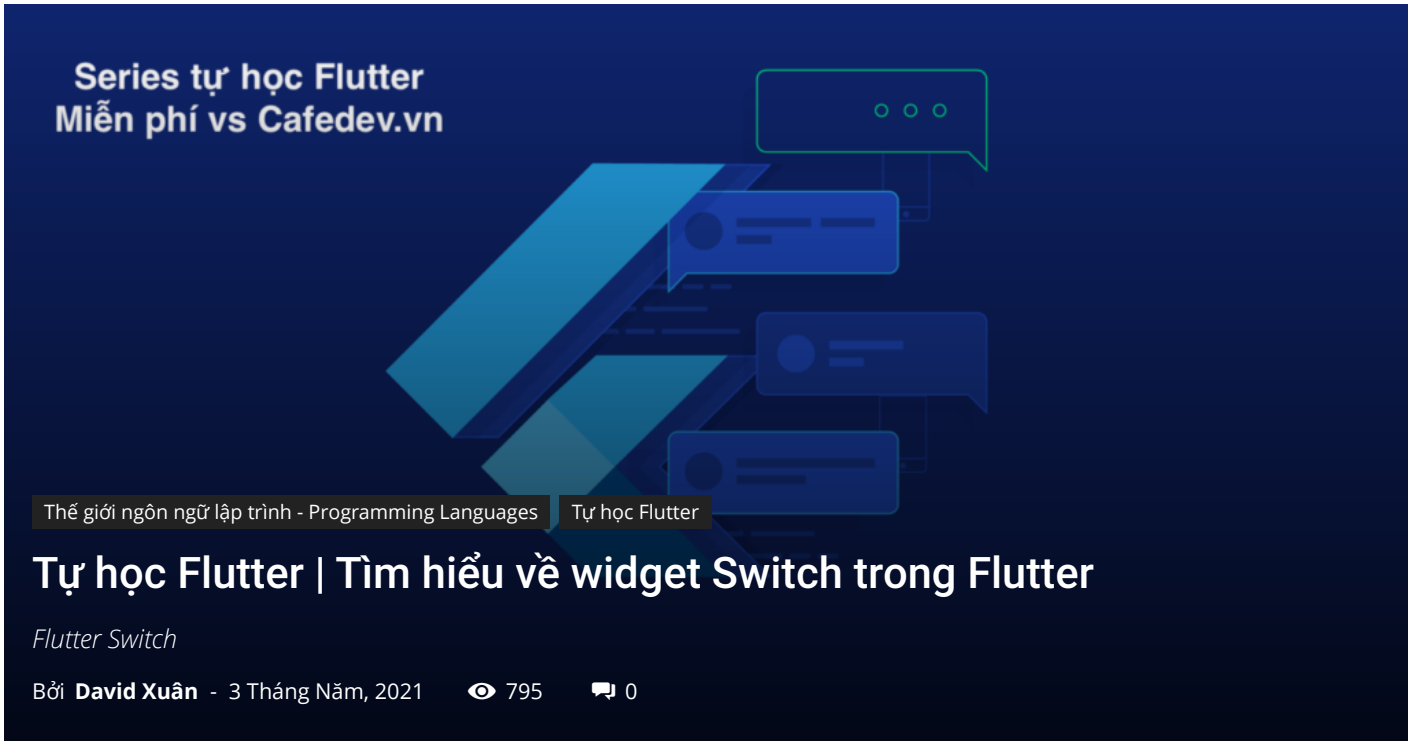
Tự học Flutter | Tìm hiểu về Navigation and Routing trong Flutter

Bài tiếp theo

Tự học Flutter | Tìm hiểu về Flutter Packages

David Xuân

https://cafedev.vn/



Tự học Flutter | Tìm hiểu về widget Switch trong Flutter

Flutter Switch

Bởi **David Xuân** - 3 Tháng Năm, 2021 795 0

Công tắc(Switch) là một phần tử giao diện người dùng hai trạng thái được sử dụng để chuyển đổi giữa các trạng thái **BẬT (Đã kiểm tra) hoặc TẮT (Bỏ chọn)** . Thông thường, nó là một nút có thanh trượt ngón tay cái, nơi người dùng có thể kéo qua lại để chọn một tùy chọn ở dạng BẬT hoặc TẮT. Hoạt động của nó tương tự như công tắc điện trong nhà.

Trong Flutter , công tắc là một widget dùng để chọn giữa hai tùy chọn BẬT hoặc TẮT. Nó không tự duy trì trạng thái. Để duy trì các trạng thái, nó sẽ gọi thuộc tính **onChanged** . Nếu giá trị trả về bởi thuộc tính này là **true** , thì công tắc sẽ BẬT và sai khi nó TẮT. Khi thuộc tính này là null, widget chuyển đổi sẽ bị vô hiệu hóa. Trong bài viết này, chúng ta sẽ tìm hiểu cách sử dụng widget chuyển đổi trong ứng dụng Flutter.

Nội dung chính

1. Thuộc tính của Switch Widget

Một số thuộc tính cần thiết của widget chuyển đổi được đưa ra dưới đây:

Thuộc tính	**Mô tả**
onChanged	Nó sẽ được gọi bất cứ khi nào người dùng chạm vào công tắc.
value	Nó chứa giá trị Boolean true hoặc false để kiểm soát xem chức năng của công tắc đang BẬT hay TẮT.
activeColor	Nó được sử dụng để chỉ định màu của công tắc bóng tròn khi nó BẬT.
activeTrackColor	Nó chỉ định màu thanh chuyển hướng.
inactiveThubmColor	Nó được sử dụng để chỉ định màu của công tắc bóng tròn khi nó TẮT.
inactiveTrackColor	Nó chỉ định màu thanh công tắc khi nó TẮT.
dragStartBehavior	Nó đã xử lý hành vi bắt đầu kéo. Nếu chúng ta đặt nó là DragStartBehavior.start, thì thao tác kéo sẽ di chuyển công tắc từ bật sang tắt.

Thí dụ

Trong ứng dụng này, chúng tôi đã xác định một **widget chuyển đổi** . Mỗi khi chúng tôi bật tắt widget chuyển đổi, thuộc tính onChanged được gọi với trạng thái mới của công tắc dưới dạng giá trị. Để lưu trữ trạng thái chuyển đổi, chúng tôi đã xác định một **biến boolean isSwitched** có thể được hiển thị trong đoạn mã dưới đây.

Mở IDE bạn đang sử dụng và tạo một ứng dụng Flutter. Tiếp theo, mở thư mục **lib** và thay thế **main.dart** bằng đoạn code sau.

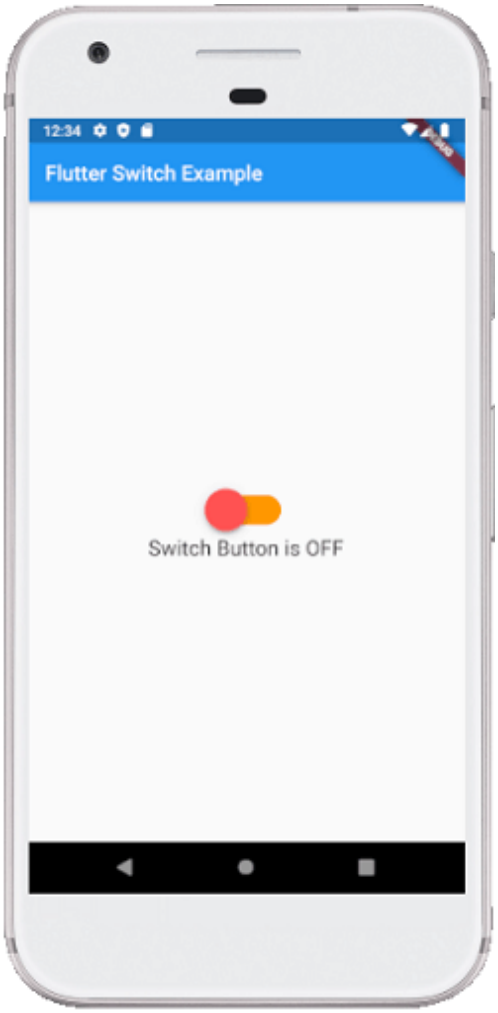
```
import 'package:flutter/material.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
        home: Scaffold(
            appBar: AppBar(
              backgroundColor: Colors.blue,
              title: Text("Flutter Switch Example"),
```

```
            ),
            body: Center(
                  child: SwitchScreen()
            ),
        )
    );
  }
}

class SwitchScreen extends StatefulWidget {
  @override
  SwitchClass createState() => new SwitchClass();
}

class SwitchClass extends State {
  bool isSwitched = false;
  var textValue = 'Switch is OFF';

  void toggleSwitch(bool value) {

    if(isSwitched == false)
    {
      setState(() {
        isSwitched = true;
        textValue = 'Switch Button is ON';
      });
      print('Switch Button is ON');
    }
    else
    {
      setState(() {
        isSwitched = false;
        textValue = 'Switch Button is OFF';
      });
      print('Switch Button is OFF');
    }
  }
  @override
  Widget build(BuildContext context) {
    return Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children:[ Transform.scale(
            scale: 2,
            child: Switch(
              onChanged: toggleSwitch,
```

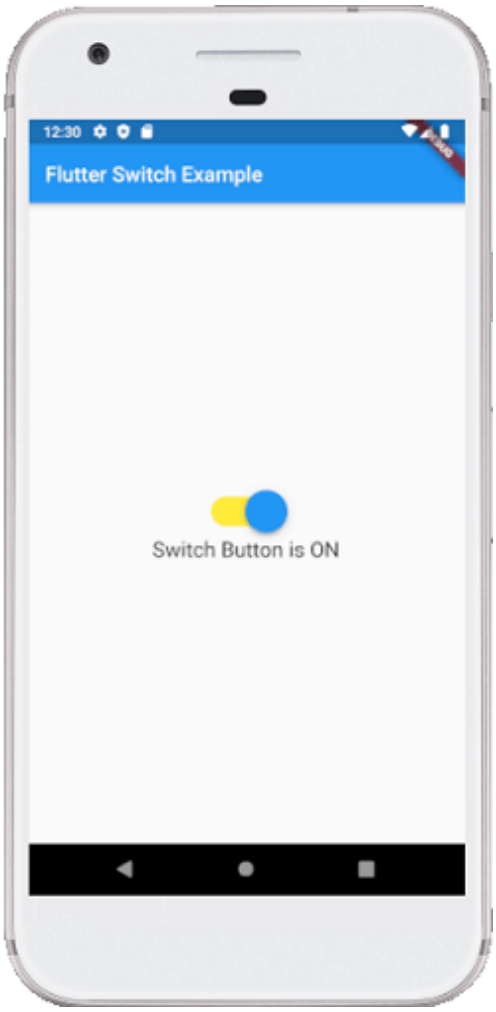
```
              value: isSwitched,
              activeColor: Colors.blue,
              activeTrackColor: Colors.yellow,
              inactiveThumbColor: Colors.redAccent,
              inactiveTrackColor: Colors.orange,
            )
          ),
          Text('$textValue', style: TextStyle(fontSize: 20),)
        ]);
  }
}
```

Đầu ra:

Khi chúng tôi chạy ứng dụng trong trình giả lập hoặc thiết bị, chúng tôi sẽ nhận được giao diện người dùng tương tự như ảnh chụp màn hình sau



Nếu chúng ta nhấn vào công tắc, nó sẽ thay đổi trạng thái của chúng từ TẮT sang BẬT. Xem ảnh chụp màn hình bên dưới:



2. Làm cách nào để tùy chỉnh nút Switch trong Flutter?

Flutter cũng cho phép người dùng tùy chỉnh nút chuyển đổi của họ. Tùy chỉnh làm cho giao diện người dùng tương tác hơn. Chúng tôi có thể làm điều này bằng cách thêm **phụ thuộc** custom_switch trong tệp **pubspec.yaml** và sau đó nhập nó vào tệp dart.

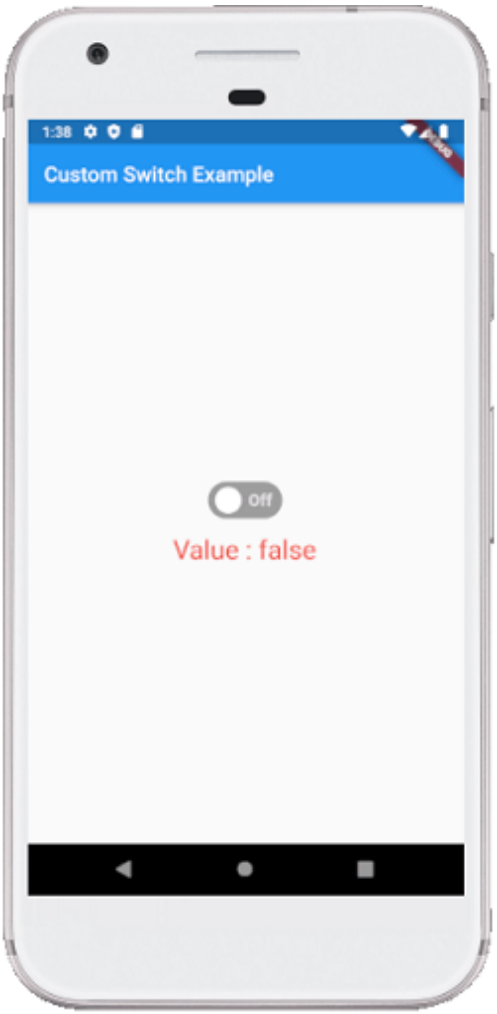
Thí dụ:

```
import 'package:flutter/material.dart';
import 'package:custom_switch/custom_switch.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
        home: Scaffold(
            appBar: AppBar(
```

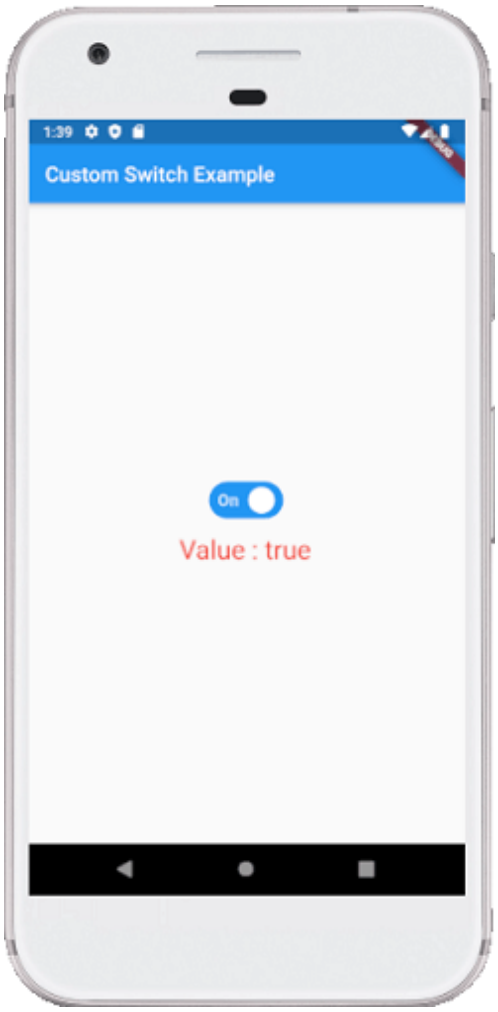
```
                backgroundColor: Colors.blue,
                title: Text("Custom Switch Example"),
              ),
              body: Center(
                    child: SwitchScreen()
              ),
          )
      );
    }
  }

  class SwitchScreen extends StatefulWidget {
    @override
    SwitchClass createState() => new SwitchClass();
  }

  class SwitchClass extends State {
    bool isSwitched = false;
    @override
    Widget build(BuildContext context) {
      return Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children:<Widget>[
              CustomSwitch(
                value: isSwitched,
                activeColor: Colors.blue,
                onChanged: (value) {
                  print("VALUE : $value");
                  setState(() {
                    isSwitched = value;
                  });
                },
              ),
            SizedBox(height: 15.0,),
            Text('Value : $isSwitched', style: TextStyle(color: Colors
                fontSize: 25.0),)
          ]);
      }
  }
```

Đầu ra:

Khi chúng tôi chạy ứng dụng trong trình giả lập hoặc thiết bị, chúng tôi sẽ nhận được giao diện người dùng tương tự như ảnh chụp màn hình sau:



Nếu chúng ta nhấn vào công tắc, nó sẽ thay đổi trạng thái của chúng từ TẮT sang BẬT. Xem ảnh chụp màn hình bên dưới:



Cài ứng dụng cafedev để dễ dàng cập nhật tin và học lập trình mọi lúc mọi nơi tại đây.

Tài liệu từ cafedev:

- Full series tự học Flutter từ cơ bản tới nâng cao tại đây nha.
- **Các nguồn kiến thức MIỄN PHÍ VÔ GIÁ từ cafedev tại đây**

Nếu bạn thấy hay và hữu ích, bạn có thể tham gia các kênh sau của cafedev để nhận được nhiều hơn nữa:

- **Group Facebook**
- **Fanpage**
- **Youtube**
- **Instagram**
- **Twitter**

- **Linkedin**
- **Pinterest**
- **Trang chủ**

Chào thân ái và quyết thắng!

Đăng ký kênh youtube để ủng hộ Cafedev nha các bạn, Thanks you!

Bài trước

Tự học Flutter | Tìm hiểu về widget Slider trong Flutter

Bài tiếp theo

Tự học Flutter | Tìm hiểu về widget Charts trong Flutter

David Xuân

https://cafedev.vn/

Thế giới ngôn ngữ lập trình - Programming Languages | Tự học Flutter

Tự học Flutter | Tìm hiểu về widget Table trong Flutter

Flutter Table

Bởi **David Xuân** - 3 Tháng Năm, 2021 807 0

Bảng(table) cho phép người dùng sắp xếp dữ liệu theo hàng và cột. Nó được sử dụng để lưu trữ và hiển thị dữ liệu của chúng ta ở định dạng có cấu trúc, giúp chúng ta so sánh các cặp giá trị có liên quan một cách dễ dàng.

Flutter cũng cho phép người dùng tạo bố cục bảng trong ứng dụng di động. Chúng ta có thể tạo một bảng trong Flutter bằng **widget Table** sử dụng thuật toán bố trí bảng cho các bảng con của nó. widget này có một số thuộc tính để nâng cao hoặc sửa đổi bố cục bảng. Các thuộc tính này là: border, children, columnWidths, textDirection, textBaseline, v.v.

Nội dung chính

1. Khi chúng ta sử dụng widget Bảng?

Một widget bảng có thể được sử dụng khi chúng ta muốn lưu trữ nhiều hàng có cùng chiều rộng cột và mỗi cột (bảng) chứa dữ liệu bằng nhau. Flutter cung cấp một cách

tiếp cận khác cho cùng một cách sử dụng **widget GridView** .

Để tạo một bảng, chúng ta phải sử dụng những thứ sau:

1. Đầu tiên, chúng ta cần thêm một **widget Table** trong nội dung.
2. Tiếp theo, chúng ta phải thêm **(các) TableRow** trong phần **child** của widget bảng. Vì widget bảng có nhiều hàng, vì vậy chúng ta sử dụng **child**, không phải **child**.
3. Cuối cùng, chúng ta cần thêm **(các)** TableCell trong phần **child** của widget TableRow. Bây giờ, chúng ta có thể viết bất kỳ widget con nào ở nơi này giống như chúng ta sẽ sử dụng **widget Text**.

```
TableRow(children: [
  TableCell(child: Text('javatpoint')),
  TableCell(
    child: Text('Flutter'),
  ),
  TableCell(child: Text('Android')),
  TableCell(child: Text('MySQL')),
]),
```

Trong khi sử dụng widget này, chúng ta phải biết các quy tắc sau:

1. widget này tự động quyết định chiều rộng cột, được chia đều cho các TableCell. Nếu nó không bằng nhau, chúng ta sẽ gặp lỗi cho biết mọi TableRow trong bảng phải có cùng số con để mọi ô đều được lấp đầy. Nếu không, bảng sẽ chứa các lỗ.
2. Mỗi hàng có cùng chiều cao, sẽ bằng chiều cao nhất của TableCell.
3. Các con của một bảng chỉ có thể có các widget TableRow.

Hãy để chúng ta hiểu điều đó với sự trợ giúp của một ví dụ dưới đây, nơi chúng ta cố gắng đề cập đến từng điều liên quan đến widget con này:

```
import 'package:flutter/material.dart';

void main() {runApp(MyApp());}

class MyApp extends StatefulWidget {
  @override
  _TableExample createState() => _TableExample();
```

```
}

class _TableExample extends State<MyApp> {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
          appBar: AppBar(
            title: Text('Flutter Table Example'),
          ),
          body: Center(
              child: Column(children: <Widget>[
                Container(
                  margin: EdgeInsets.all(20),
                  child: Table(
                    defaultColumnWidth: FixedColumnWidth(120.0),
                    border: TableBorder.all(
                        color: Colors.black,
                        style: BorderStyle.solid,
                        width: 2),
                    children: [
                      TableRow( children: [
                        Column(children:[Text('Website', style: Text
                        Column(children:[Text('Tutorial', style: Tex
                        Column(children:[Text('Review', style: TextS
                      ]),
                      TableRow( children: [
                        Column(children:[Text('Javatpoint')]),
                        Column(children:[Text('Flutter')]),
                        Column(children:[Text('5*')]),
                      ]),
                      TableRow( children: [
                        Column(children:[Text('Javatpoint')]),
                        Column(children:[Text('MySQL')]),
                        Column(children:[Text('5*')]),
                      ]),
                      TableRow( children: [
                        Column(children:[Text('Javatpoint')]),
                        Column(children:[Text('ReactJS')]),
                        Column(children:[Text('5*')]),
                      ]),
                    ],
                  ),
                ),
              ])
```

```
54                )),
55          );
56      }
    }
```

Đầu ra:

Khi chúng ta chạy ứng dụng trong trình giả lập hoặc thiết bị, chúng ta sẽ thấy ảnh chụp màn hình bên dưới:

2. Flutter DataTable

Flutter cũng cho phép chúng ta một widget khác để tạo một bảng trong ứng dụng của chúng ta có tên là widget **DataTable** . Nó là một bảng dữ liệu thiết kế material design, nơi chúng ta có thể hiển thị dữ liệu với **các nhãn cột và hàng** . widget này tự động điều chỉnh cột của bảng theo dữ liệu ô. Việc hiển thị dữ liệu bằng widget này rất tốn kém vì ở đây, tất cả dữ liệu phải được đo hai lần. Đầu tiên, nó đo kích thước cho từng

cột và thứ hai, nó thực sự đưa ra bảng. Vì vậy, chúng ta phải đảm bảo rằng widget con này chỉ có thể được sử dụng khi chúng ta có ít hàng hơn.

widget DataTable được lưu trữ thông tin bằng cách sử dụng thuộc tính **cột và hàng** . Thuộc tính cột chứa dữ liệu bằng cách sử dụng một mảng **DataColumn** và thuộc tính hàng chứa thông tin bằng cách sử dụng một mảng **DataRow** . DataRow có phụ thuộc **tế bào** mà phải mất một mảng của **DataCell** . DataColumn có một **nhãn** thuộc tính phụ nhận các widget làm **giá trị**. Chúng ta cũng có thể cung cấp Văn bản, Hình ảnh, Biểu tượng hoặc bất kỳ widget nào khác trong DataTable.

Sau đây là cú pháp của DataTable:

```
DataTable(
  columns: [
    DataColumn(label: ),
    DataColumn(label: )),
  ],
  rows: [
    DataRow(cells: [
      DataCell( ),
      DataCell( ),
      ...
      DataCell( ),
    ]),
    DataRow(cells: [
      DataCell( ),
      DataCell( ),
      ...
      DataCell( ),
    ]),
  ],
),
```

Thí dụ

Hãy để chúng ta hiểu cách sử dụng DataTable trong ứng dụng Flutter. Ở đây, chúng ta sẽ xác định một bảng dữ liệu đơn giản có **ba cột và bốn hàng** :

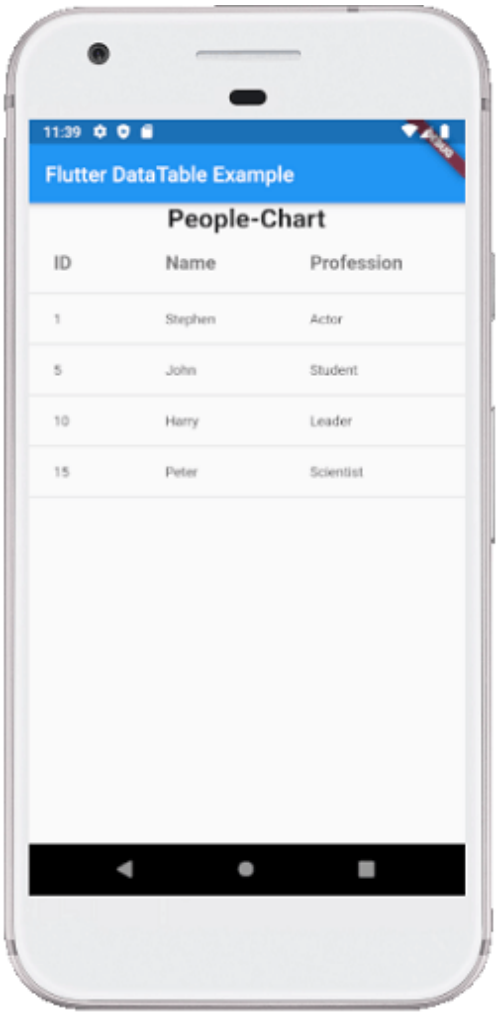
```
import 'package:flutter/material.dart';
```

```
void main() {runApp(MyApp());}

class MyApp extends StatefulWidget {
  @override
  _DataTableExample createState() => _DataTableExample();
}

class _DataTableExample extends State<MyApp> {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
          appBar: AppBar(
            title: Text('Flutter DataTable Example'),
          ),
          body: ListView(children: <Widget>[
            Center(
                child: Text(
                  'People-Chart',
                  style: TextStyle(fontSize: 25, fontWeight: FontWeight.b
                )),
            DataTable(
              columns: [
                DataColumn(label: Text(
                    'ID',
                    style: TextStyle(fontSize: 18, fontWeight: FontWeight
                )),
                DataColumn(label: Text(
                    'Name',
                    style: TextStyle(fontSize: 18, fontWeight: FontWeight
                )),
                DataColumn(label: Text(
                    'Profession',
                    style: TextStyle(fontSize: 18, fontWeight: FontWeight
                )),
              ],
              rows: [
                DataRow(cells: [
                  DataCell(Text('1')),
                  DataCell(Text('Stephen')),
                  DataCell(Text('Actor')),
                ]),
                DataRow(cells: [
                  DataCell(Text('5')),
                  DataCell(Text('John')),
```

```
                DataCell(Text('Student')),
              ]),
              DataRow(cells: [
                DataCell(Text('10')),
                DataCell(Text('Harry')),
                DataCell(Text('Leader')),
              ]),
              DataRow(cells: [
                DataCell(Text('15')),
                DataCell(Text('Peter')),
                DataCell(Text('Scientist')),
              ]),
            ],
          ),
        ])
      ),
    );
  }
}
```

Đầu ra:

Khi chúng ta chạy ứng dụng trong trình giả lập hoặc thiết bị, chúng ta sẽ thấy ảnh chụp màn hình bên dưới:



Cài ứng dụng cafedev để dễ dàng cập nhật tin và học lập trình mọi lúc mọi nơi tại đây.

Tài liệu từ cafedev:

- Full series tự học Flutter từ cơ bản tới nâng cao tại đây nha.
- **Các nguồn kiến thức MIỄN PHÍ VÔ GIÁ từ cafedev tại đây**

Nếu bạn thấy hay và hữu ích, bạn có thể tham gia các kênh sau của cafedev để nhận được nhiều hơn nữa:

- **Group Facebook**
- **Fanpage**
- **Youtube**
- **Instagram**
- **Twitter**

- **Linkedin**
- **Pinterest**
- **Trang chủ**

Chào thân ái và quyết thắng!

Đăng ký kênh youtube để ủng hộ Cafedev nha các bạn, Thanks you!

Bài trước

Tự học Flutter | Tìm hiểu về widgets theme trong Flutter

Bài tiếp theo

Tự học Flutter | Tìm hiểu về animation trong Flutter

David Xuân

https://cafedev.vn/

Thế giới ngôn ngữ lập trình - Programming Languages | Tự học Flutter

Tự học Flutter | Tìm hiểu về widget Bottom Navigation Bar trong Flutter

Flutter Bottom Navigation Bar

Bởi **David Xuân** - 3 Tháng Năm, 2021 823 0

Thanh Điều hướng(Bottom Navigation Bar) dưới cùng đã trở nên phổ biến trong vài năm gần đây để điều hướng giữa các giao diện người dùng khác nhau. Nhiều nhà phát triển sử dụng điều hướng dưới cùng vì hầu hết ứng dụng hiện có sẵn bằng cách sử dụng widget này để điều hướng giữa các màn hình khác nhau.

Thanh điều hướng dưới cùng trong Flutter **có thể chứa nhiều mục** như nhãn văn bản, biểu tượng hoặc cả hai. Nó cho phép người dùng điều hướng giữa các chế độ xem cấp cao nhất của ứng dụng một cách nhanh chóng. Nếu chúng ta đang sử dụng màn hình lớn hơn, tốt hơn nên sử dụng **thanh điều hướng bên** .

Trong ứng dụng Flutter , chúng tôi thường đặt thanh điều hướng phía dưới cùng với widget **Scaffold**. widget con **Scaffold** cung cấp đối số **Scaffold.bottomNavigationBar** để đặt thanh điều hướng dưới cùng. Cần lưu ý rằng chỉ thêm BottomNavigationBar sẽ không hiển thị các mục điều hướng. Bắt buộc phải đặt thuộc tính **BottomNavigationItems** cho các mục chấp nhận danh sách các widget con BottomNavigationItems.

Hãy nhớ những điểm chính sau khi thêm các mục vào thanh điều hướng dưới cùng:

- Chúng tôi chỉ có thể hiển thị một số lượng nhỏ widget con trong điều hướng phía dưới, có thể là 2 đến 5.
- Nó phải có ít nhất hai mục điều hướng dưới cùng. Nếu không, chúng tôi sẽ gặp lỗi.
- Bắt buộc phải có thuộc tính biểu tượng và tiêu đề, và chúng ta cần đặt các widget có liên quan cho chúng.

Nội dung chính

1. Thuộc tính của widget con BottomNavigationBar

Sau đây là các thuộc tính được sử dụng với widget thanh điều hướng dưới cùng:

items: Nó xác định danh sách để hiển thị trong thanh điều hướng dưới cùng. Nó sử dụng đối số BottomNavigationBarItem có chứa các thuộc tính sup được đưa ra bên dưới:

```
const BottomNavigationBarItem({
    @required this.icon,
    this.title,
    Widget activeIcon,
    this.backgroundColor,
  })
```

currentIndex: Nó xác định mục thanh điều hướng dưới cùng đang hoạt động hiện tại trên màn hình.

onTap: Nó được gọi khi chúng ta chạm vào một trong các mục trên màn hình.

iconSize: Nó được sử dụng để chỉ định kích thước của tất cả các biểu tượng mục điều hướng phía dưới.

fixedColor: Nó được sử dụng để đặt màu của mục đã chọn. Nếu chúng ta chưa đặt màu cho biểu tượng hoặc tiêu đề, nó sẽ được hiển thị.

type: Nó xác định bố cục và hành vi của thanh điều hướng dưới cùng. Nó hoạt động theo hai cách khác nhau, đó là: **fixed** và shifting. Nếu nó là null, nó sẽ sử dụng fixed. Nếu không, nó sẽ sử dụng tính năng chuyển đổi nơi chúng ta có thể xem hoạt ảnh khi chúng ta nhấp vào một nút.

2. Thí dụ:

Hãy để chúng tôi hiểu cách tạo thanh điều hướng dưới cùng trong ứng dụng Flutter với sự trợ giúp của ví dụ. Vì vậy, hãy mở studio android và tạo ứng dụng Flutter. Kế tiếp. Mở tệp **main.dart** và xóa code của nó bằng đoạn mã dưới đây:

```
import 'package:flutter/material.dart';

void main() => runApp(MyApp());

/// This Widget is the main application widget.
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: MyNavigationBar (),
    );
  }
}

class MyNavigationBar extends StatefulWidget {
  MyNavigationBar ({Key key}) : super(key: key);

  @override
  _MyNavigationBarState createState() => _MyNavigationBarState();
}

class _MyNavigationBarState extends State<MyNavigationBar > {
  int _selectedIndex = 0;
  static const List<Widget> _widgetOptions = <Widget>[
    Text('Home Page', style: TextStyle(fontSize: 35, fontWeight: Fon
    Text('Search Page', style: TextStyle(fontSize: 35, fontWeight: F
    Text('Profile Page', style: TextStyle(fontSize: 35, fontWeight:
  ];

  void _onItemTapped(int index) {
    setState(() {
      _selectedIndex = index;
```

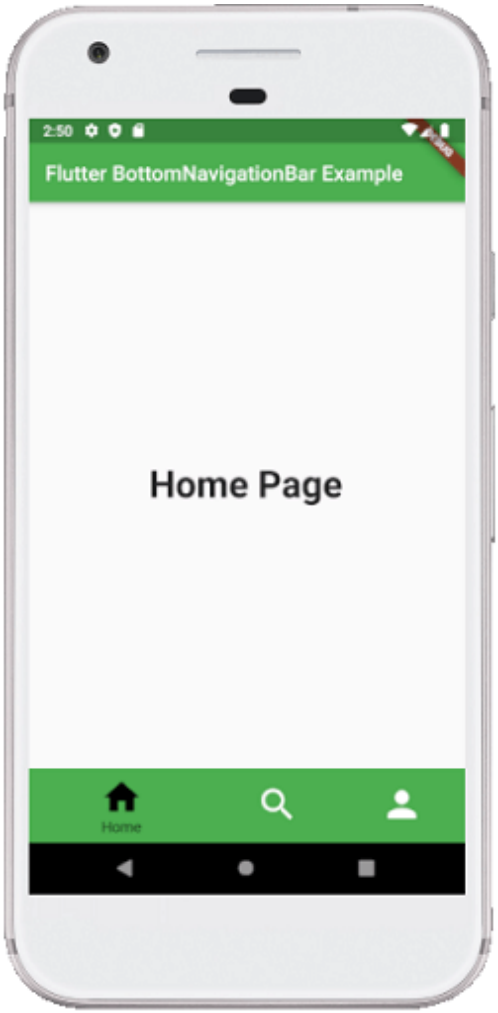
```
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Flutter BottomNavigationBar Example'),
          backgroundColor: Colors.green
      ),
      body: Center(
        child: _widgetOptions.elementAt(_selectedIndex),
      ),
      bottomNavigationBar: BottomNavigationBar(
        items: const <BottomNavigationBarItem>[
          BottomNavigationBarItem(
            icon: Icon(Icons.home),
            title: Text('Home'),
            backgroundColor: Colors.green
          ),
          BottomNavigationBarItem(
            icon: Icon(Icons.search),
            title: Text('Search'),
            backgroundColor: Colors.yellow
          ),
          BottomNavigationBarItem(
            icon: Icon(Icons.person),
            title: Text('Profile'),
            backgroundColor: Colors.blue,
          ),
        ],
        type: BottomNavigationBarType.shifting,
        currentIndex: _selectedIndex,
        selectedItemColor: Colors.black,
        iconSize: 40,
        onTap: _onItemTapped,
        elevation: 5
      ),
    );
  }
}
```

Trong đoạn mã trên, chúng ta đã sử dụng BottomNavigationBar trong một widget con đầu đàn. Điều hướng này chứa **ba widget con BottomNavigationBarItem** . Ở đây,

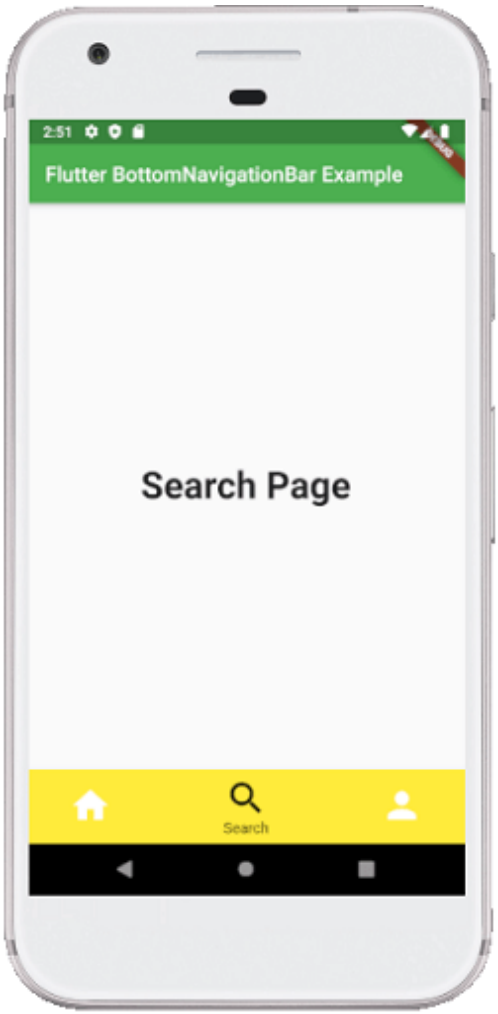
chúng tôi đã đặt **currentIndex** thành 0 để chọn một mục có **màu xanh lá cây** . Hàm **onTap ()** được sử dụng để thay đổi chỉ mục của mục đã chọn và sau đó hiển thị một thông báo tương ứng.

Đầu ra:

Khi chúng tôi chạy ứng dụng, chúng ta sẽ nhận được giao diện người dùng tương tự như ảnh chụp màn hình bên dưới:



Khi chúng ta nhấp vào **biểu tượng tìm kiếm** ở thanh điều hướng dưới cùng, nó sẽ đưa ra màn hình bên dưới.



Cài ứng dụng cafedev để dễ dàng cập nhật tin và học lập trình mọi lúc mọi nơi tại đây.

Tài liệu từ cafedev:

- Full series tự học Flutter từ cơ bản tới nâng cao tại đây nha.
- **Các nguồn kiến thức MIỄN PHÍ VÔ GIÁ từ cafedev tại đây**

Nếu bạn thấy hay và hữu ích, bạn có thể tham gia các kênh sau của cafedev để nhận được nhiều hơn nữa:

- **Group Facebook**
- **Fanpage**
- **Youtube**
- **Instagram**
- **Twitter**

- **Linkedin**
- **Pinterest**
- **Trang chủ**

Chào thân ái và quyết thắng!

Đăng ký kênh youtube để ủng hộ Cafedev nha các bạn, Thanks you!

Bài trước

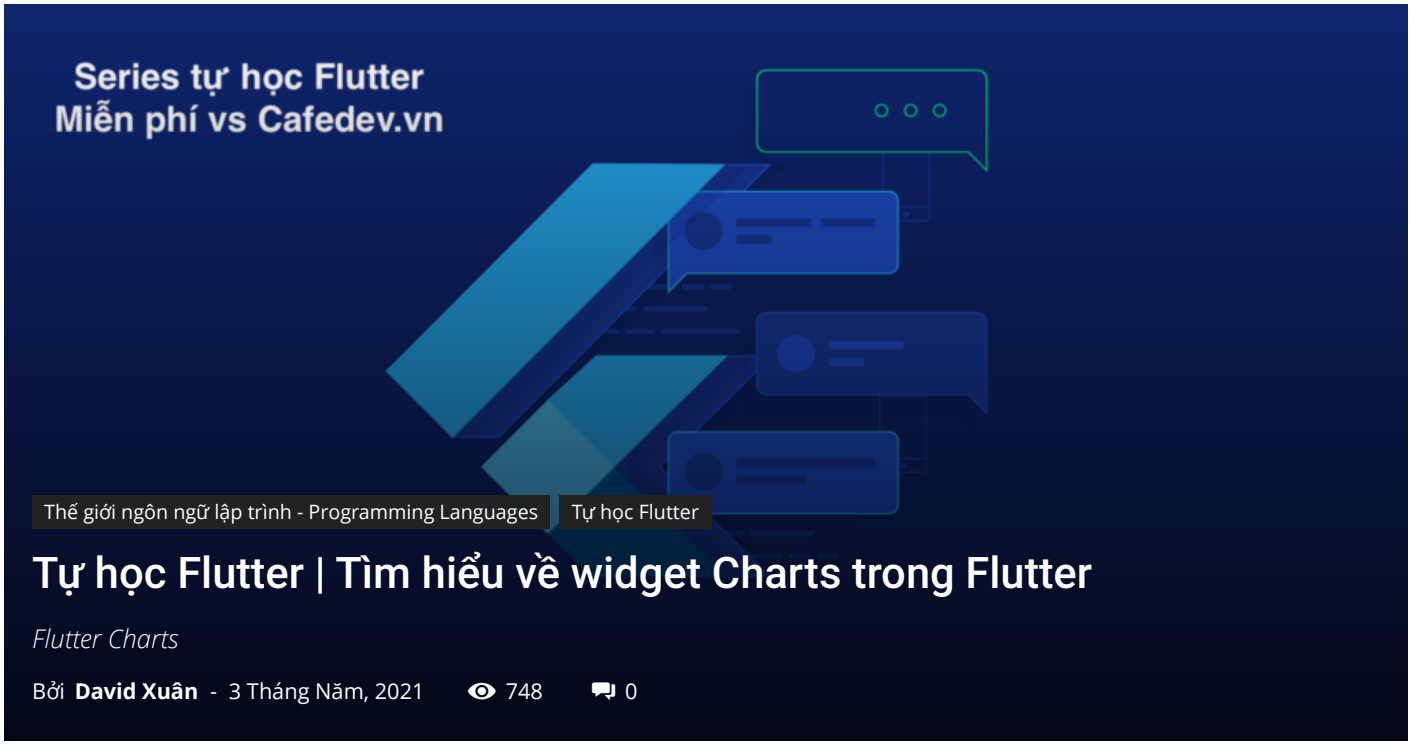
Tự học Flutter | Tìm hiểu về widget Charts trong Flutter

Bài tiếp theo

Tự học Flutter | Tìm hiểu về widgets theme trong Flutter

David Xuân

https://cafedev.vn/



Tự học Flutter | Tìm hiểu về widget Charts trong Flutter

Flutter Charts

Bởi **David Xuân** - 3 Tháng Năm, 2021 748 0

Biểu đồ là một biểu diễn đồ họa của dữ liệu trong đó dữ liệu được biểu thị bằng một **ký hiệu** như đường thẳng, thanh, hình tròn, v.v. Trong Flutter, biểu đồ hoạt động giống như một biểu đồ bình thường. Chúng ta sử dụng một biểu đồ trong Flutter để **biểu diễn dữ liệu theo cách đồ họa** cho phép người dùng hiểu chúng một cách đơn giản. Chúng ta cũng có thể vẽ một biểu đồ để biểu thị sự tăng và giảm của các giá trị của chúng ta. Biểu đồ có thể dễ dàng đọc dữ liệu và giúp chúng ta biết hiệu suất hàng tháng hoặc hàng năm bất cứ khi nào chúng ta cần.

Nội dung chính

1. Các loại biểu đồ được hỗ trợ trong Flutter

Flutter chủ yếu hỗ trợ ba loại biểu đồ và mỗi biểu đồ đi kèm với một số tùy chọn cấu hình. Sau đây là biểu đồ được sử dụng trong ứng dụng Flutter:

1. Biểu đồ đường

2. Biểu đồ cột

3. Biểu đồ bánh và bánh donut

1.1 Biểu đồ đường

Biểu đồ đường là biểu đồ sử dụng các đường để kết nối các điểm dữ liệu riêng lẻ. Nó hiển thị thông tin trong một loạt các điểm dữ liệu. Nó chủ yếu được sử dụng để theo dõi các thay đổi trong một khoảng thời gian ngắn và dài.

Chúng ta có thể sử dụng nó như sau:

```
LineChart(
  LineChartData(
    // write your logic
  ),
);
```

1.2 Biểu đồ cột

Đây là một biểu đồ đại diện cho dữ liệu phân loại với các thanh hình chữ nhật. Nó có thể nằm ngang hoặc dọc.

Chúng ta có thể sử dụng nó như sau:

```
BarChart(
  BarChartData(
    // write your logic
  ),
);
```

1.3 Biểu đồ bánh hoặc bánh donut

Nó là một đồ thị hiển thị thông tin dưới dạng đồ thị hình tròn. Trong biểu đồ này, vòng tròn được chia thành các ngành và mỗi phần hiển thị dữ liệu phần trăm hoặc tỷ lệ.

Chúng ta có thể sử dụng nó như sau:

```
PieChart(
  PieChartData(
    // write your logic
  ),
);
```

Hãy để chúng tôi hiểu nó với sự trợ giúp của một ví dụ.

1.4 Thí dụ

Mở IDE và tạo dự án Flutter mới. Tiếp theo, mở dự án, điều hướng đến thư mục lib và mở tệp **pubspec.yaml** . Trong tệp này, chúng ta cần thêm sự phụ thuộc của biểu đồ. Flutter cung cấp một số phụ thuộc biểu đồ và ở đây chúng ta sẽ sử dụng **phụ thuộc fl_chart** . Vì vậy, thêm nó như dưới đây:

```
dependencies:
  flutter:
    sdk: flutter
  fl_chart: ^0.10.1
```

Sau khi thêm phần phụ thuộc, hãy nhấp vào liên kết **nhận gói** được hiển thị ở góc trên cùng bên trái của màn hình. Bây giờ, hãy mở tệp **main.dart** và thay thế nó bằng đoạn code dưới đây:

```
import 'package:flutter/material.dart';
import 'package:fl_chart/fl_chart.dart';

void main() => runApp(MyApp());

/// This Widget is the main application widget.
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: HomePage(),
    );
  }
}

class HomePage extends StatelessWidget {
  @override
```

```
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Flutter Chart Example'),
          backgroundColor: Colors.green
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            LineCharts(),
            Padding(
              padding: const EdgeInsets.all(16.0),
              child: Text(
                "Traffic Source Chart",
                  style: TextStyle(
                      fontSize: 20,
                      color: Colors.purple,
                      fontWeight: FontWeight.w700,
                      fontStyle: FontStyle.italic
                  )
              )
            ),
          ],
        ),
      ),
    );
  }
}

class LineCharts extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    const cutOffYValue = 0.0;
    const yearTextStyle =
    TextStyle(fontSize: 12, color: Colors.black);

    return SizedBox(
      width: 360,
      height: 250,
      child: LineChart(
        LineChartData(
          lineTouchData: LineTouchData(enabled: false),
          lineBarsData: [
            LineChartBarData(
```

```
          spots: [
            FlSpot(0, 1),
            FlSpot(1, 1),
            FlSpot(2, 3),
            FlSpot(3, 4),
            FlSpot(3, 5),
            FlSpot(4, 4)
          ],
          isCurved: true,
          barWidth: 2,
          colors: [
            Colors.black,
          ],
          belowBarData: BarAreaData(
            show: true,
            colors: [Colors.lightBlue.withOpacity(0.5)],
            cutOffY: cutOffYValue,
            applyCutOffY: true,
          ),
          aboveBarData: BarAreaData(
            show: true,
            colors: [Colors.lightGreen.withOpacity(0.5)],
            cutOffY: cutOffYValue,
            applyCutOffY: true,
          ),
          dotData: FlDotData(
            show: false,
          ),
        ),
      ],
      minY: 0,
      titlesData: FlTitlesData(
        bottomTitles: SideTitles(
          showTitles: true,
          reservedSize: 5,
          textStyle: yearTextStyle,
          getTitles: (value) {
            switch (value.toInt()) {
              case 0:
                return '2016';
              case 1:
                return '2017';
              case 2:
                return '2018';
              case 3:
```

```
                        return '2019';
                      case 4:
                        return '2020';
                      default:
                        return '';
                    }
                  }),
              leftTitles: SideTitles(
                showTitles: true,
                getTitles: (value) {
                  return '\$ ${value + 100}';
                },
              ),
            ),
            axisTitleData: FlAxisTitleData(
                leftTitle: AxisTitle(showTitle: true, titleText: 'Valu
                bottomTitle: AxisTitle(
                    showTitle: true,
                    margin: 10,
                    titleText: 'Year',
                    textStyle: yearTextStyle,
                    textAlign: TextAlign.right)),
            gridData: FlGridData(
              show: true,
              checkToShowHorizontalLine: (double value) {
                return value == 1 || value == 2 || value == 3 || value
              },
            ),
          ),
        ),
      );
    }
  }
```

Đầu ra:

Khi chúng tôi chạy ứng dụng trong thiết bị hoặc trình giả lập, chúng ta sẽ nhận được giao diện người dùng của màn hình tương tự như ảnh chụp màn hình bên dưới:

Cài ứng dụng cafedev để dễ dàng cập nhật tin và học lập trình mọi lúc mọi nơi tại đây.

Tài liệu từ cafedev:

- Full series tự học Flutter từ cơ bản tới nâng cao tại đây nha.
- **Các nguồn kiến thức MIỄN PHÍ VÔ GIÁ từ cafedev tại đây**

Nếu bạn thấy hay và hữu ích, bạn có thể tham gia các kênh sau của cafedev để nhận được nhiều hơn nữa:

- **Group Facebook**
- **Fanpage**
- **Youtube**
- **Instagram**
- **Twitter**

- **Linkedin**
- **Pinterest**
- **Trang chủ**

Chào thân ái và quyết thắng!

Đăng ký kênh youtube để ủng hộ Cafedev nha các bạn, Thanks you!

Bài trước

Tự học Flutter | Tìm hiểu về widget Switch trong Flutter

Bài tiếp theo

Tự học Flutter | Tìm hiểu về widget Bottom Navigation Bar trong Flutter

David Xuân

https://cafedev.vn/

Tự học Flutter | Tìm hiểu về widget Alert Dialogs trong Flutter

Flutter Alert Dialogs

Bởi **David Xuân** - 21 Tháng Tư, 2021 899 0

Hộp thoại cảnh báo(Alert Dialogs) là một tính năng hữu ích thông báo cho người dùng thông tin quan trọng để đưa ra quyết định hoặc cung cấp khả năng chọn một hành động cụ thể hoặc danh sách các hành động. Đó là một hộp bật lên xuất hiện ở đầu nội dung ứng dụng và giữa màn hình. Người dùng có thể loại bỏ nó theo cách thủ công trước khi tiếp tục tương tác với ứng dụng.

Cảnh báo có thể được coi là một phương thức nổi nên được sử dụng để phản hồi nhanh như xác minh mật khẩu, thông báo ứng dụng nhỏ, v.v. Các cảnh báo rất linh hoạt và có thể được tùy chỉnh rất dễ dàng.

Trong Flutter, AlertDialog là một widget, thông báo cho người dùng về các tình huống cần xác nhận. Hộp thoại cảnh báo Flutter chứa tiêu đề tùy chọn hiển thị phía trên nội dung và danh sách các hành động được hiển thị bên dưới nội dung.

Nội dung chính

1. Thuộc tính của hộp thoại cảnh báo

Các thuộc tính chính của tiện ích AlertDialog là:

Title: Thuộc tính này cung cấp tiêu đề cho hộp AlertDialog nằm ở đầu AlertDialog. Luôn luôn tốt để giữ tiêu đề càng ngắn càng tốt để người dùng biết về việc sử dụng nó rất dễ dàng. Chúng ta có thể viết tiêu đề trong AlertDialog như sau:

```
AlertDialog(title: Text("Sample Alert Dialog"),
```

Action: Nó hiển thị bên dưới nội dung. Ví dụ: nếu cần tạo một nút để chọn có hoặc không, thì nút đó chỉ được xác định trong thuộc tính hành động. Chúng ta có thể viết một thuộc tính action trong AlertDialog như sau:

```
1  actions: <Widget>[
2      FlatButton(child: Text("Yes"),),
3      FlatButton(child: Text("No"),)
4  ],)
```

Content: Thuộc tính này xác định **Content** của widget AlertDialog. Nó là một loại văn bản, nhưng nó cũng có thể chứa bất kỳ loại widget bố cục nào. Chúng ta có thể sử dụng thuộc tính Content trong AlertDialog như sau:

```
actions: <Widget>[
    FlatButton(child: Text("Yes"),),
    FlatButton(child: Text("No"),)
],)
```

ContentPadding: Nó cung cấp phần đệm cần thiết cho nội dung bên trong tiện ích AlertDialog. Chúng ta có thể sử dụng thuộc tính ContentPadding trong AlertDialog như sau:

```
contentPadding: EdgeInsets.all(32.0),
```

Hình dạng: Thuộc tính này cung cấp hình dạng cho hộp thoại cảnh báo, chẳng hạn như đường cong, hình tròn hoặc bất kỳ hình dạng khác nào khác.

```
shape: CircleBorder(),
shape: CurveBorder(),
```

Chúng ta có thể phân loại hộp thoại cảnh báo thành nhiều loại, được đưa ra bên dưới:

1. AlertDialog cơ bản
2. Confirmation AlertDialog
3. Select AlertDialog
4. TextField AlertDialog

2. AlertDialog cơ bản

Cảnh báo này thông báo cho người dùng về thông tin mới, chẳng hạn như thay đổi trong ứng dụng, về các tính năng mới, tình huống khẩn cấp cần xác nhận hoặc như một thông báo xác nhận cho người dùng rằng hành động có thành công hay không. Ví dụ sau giải thích việc sử dụng các cảnh báo cơ bản.

Thí dụ

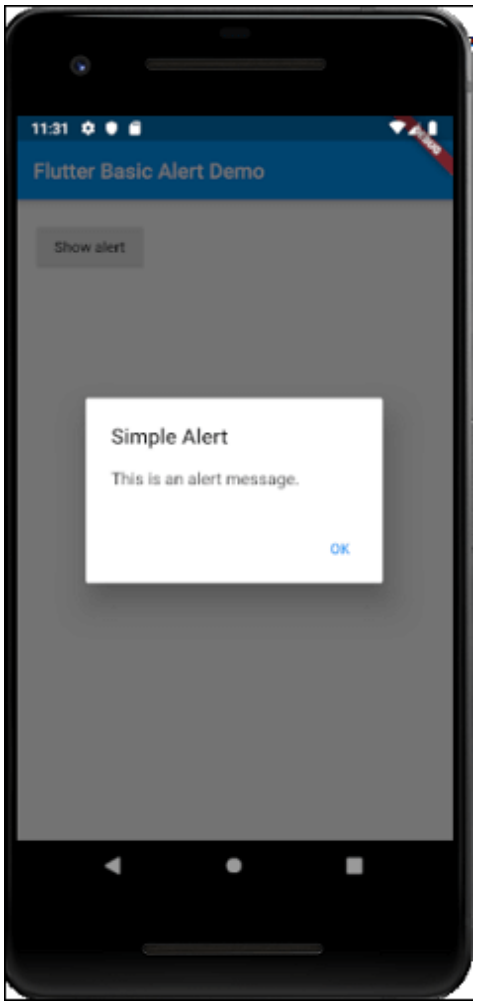
Tạo một dự án Flutter trong Android Studio và thay thế code sau bằng tệp **main.dart** . Để hiển thị một cảnh báo, bạn phải gọi **hàm showDialog()** , hàm này chứa context và hàm **itemBuilder** . Hàm itemBuilder trả về **hộp thoại** kiểu **đối tượng ,** AlertDialog.

```
import 'package:flutter/material.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final appTitle = 'Flutter Basic Alert Demo';
    return MaterialApp(
      title: appTitle,
      home: Scaffold(
        appBar: AppBar(
          title: Text(appTitle),
        ),
        body: MyAlert(),
      ),
```

```
    );
  }
}

class MyAlert extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Padding(
      padding: const EdgeInsets.all(20.0),
      child: RaisedButton(
        child: Text('Show alert'),
        onPressed: () {
          showAlertDialog(context);
        },
      ),
    );
  }
}

showAlertDialog(BuildContext context) {
  // Create button
  Widget okButton = FlatButton(
    child: Text("OK"),
    onPressed: () {
      Navigator.of(context).pop();
    },
  );

  // Create AlertDialog
  AlertDialog alert = AlertDialog(
    title: Text("Simple Alert"),
    content: Text("This is an alert message."),
    actions: [
      okButton,
    ],
  );

  // show the dialog
  showDialog(
    context: context,
    builder: (BuildContext context) {
      return alert;
    },
  );
}
```

Đầu ra

Bây giờ, hãy chạy ứng dụng, nó sẽ cho kết quả như sau. Khi bạn nhấp vào nút Hiển thị cảnh báo, bạn sẽ nhận được thông báo cảnh báo.



3. TextField AlertDialog

AlertDialog này làm cho nó có thể chấp nhận đầu vào của người dùng. Trong ví dụ sau, chúng ta sẽ thêm đầu vào trường văn bản trong hộp thoại cảnh báo. Mở tệp main.dart và chèn code sau.

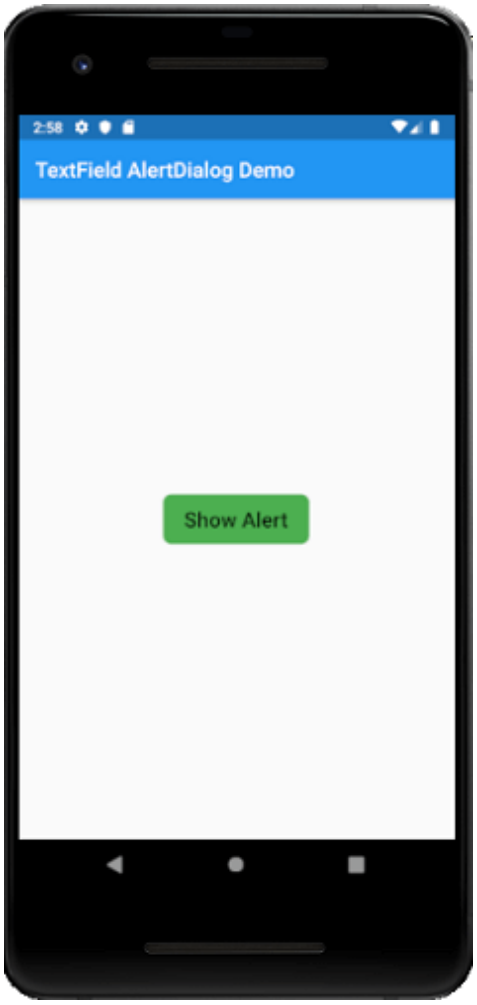
```
import 'package:flutter/material.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Alert Demo',
      debugShowCheckedModeBanner: false,
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
```

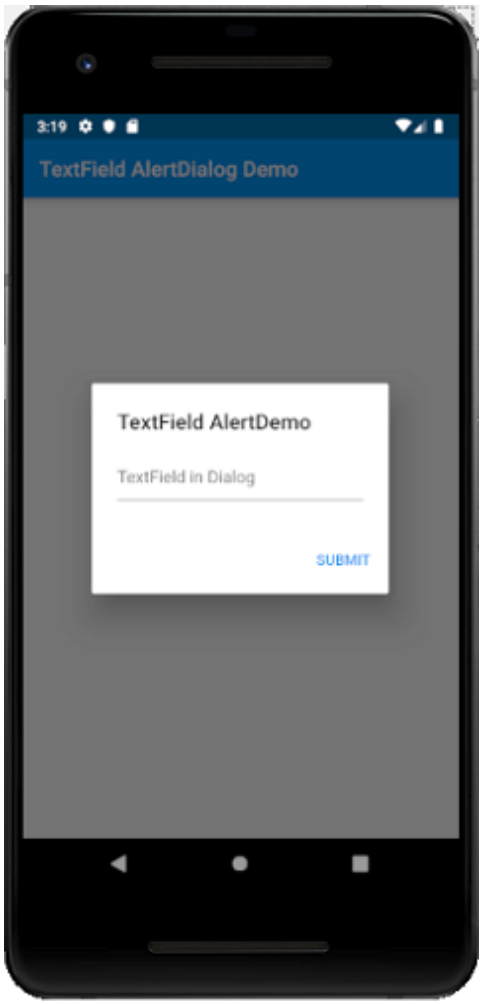
```
        //home: MyHomePage(title: 'Flutter Demo Home Page'),
        home: TextFieldAlertDialog(),
      );
    }
  }
  class TextFieldAlertDialog extends StatelessWidget {
    TextEditingController _textFieldController = TextEditingController

    _displayDialog(BuildContext context) async {
      return showDialog(
          context: context,
          builder: (context) {
            return AlertDialog(
              title: Text('TextField AlertDemo'),
              content: TextField(
                controller: _textFieldController,
                decoration: InputDecoration(hintText: "TextField in Di
              ),
              actions: <Widget>[
                new FlatButton(
                  child: new Text('SUBMIT'),
                  onPressed: () {
                    Navigator.of(context).pop();
                  },
                )
              ],
            );
          });
    }

    @override
    Widget build(BuildContext context) {
      return Scaffold(
        appBar: AppBar(
          title: Text('TextField AlertDialog Demo'),
        ),
        body: Center(
          child: FlatButton(
            child: Text(
              'Show Alert',
              style: TextStyle(fontSize: 20.0),),
              padding: EdgeInsets.fromLTRB(20.0,12.0,20.0,12.0),
              shape: RoundedRectangleBorder(
                borderRadius: BorderRadius.circular(8.0)
              ),
```

```
            color: Colors.green,
            onPressed: () => _displayDialog(context),
          ),
        ),
      );
    }
  }
```

Đầu ra

Bây giờ, hãy chạy ứng dụng, nó sẽ cho kết quả như sau. Khi bạn nhấp vào nút **Show Alert,** bạn sẽ nhận được thông báo cảnh báo nhập văn bản.





4. Confirmation AlertDialog

Hộp thoại cảnh báo xác nhận thông báo cho người dùng xác nhận một lựa chọn cụ thể trước khi chuyển tiếp trong ứng dụng. Ví dụ: khi người dùng muốn xóa hoặc xóa một số liên lạc khỏi sổ địa chỉ.

Thí dụ

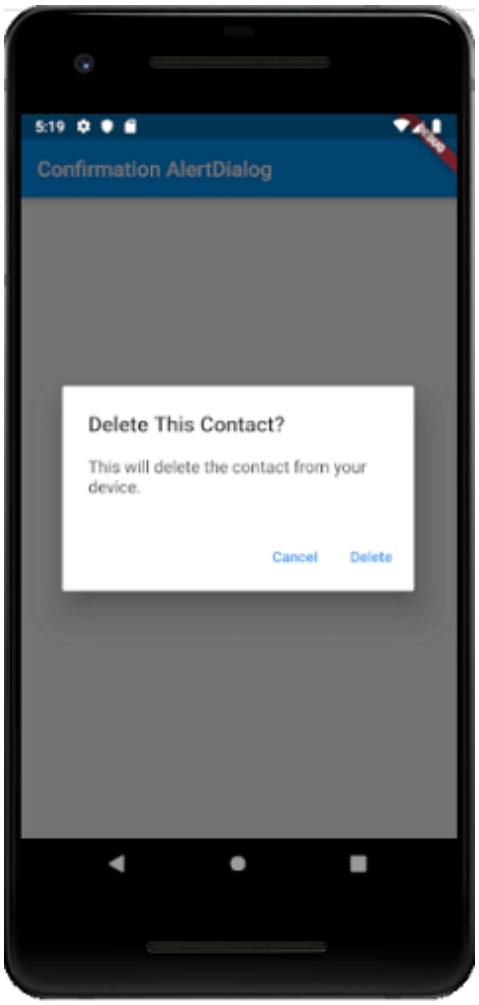
```
import 'package:flutter/material.dart';

void main() {
  runApp(new MaterialApp(home: new MyApp()));
}

class MyApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    // TODO: implement build
    return new Scaffold(
```

```
        appBar: AppBar(
          title: Text("Confirmation AlertDialog"),
        ),
        body: Center(
          child: Column(
            mainAxisAlignment: MainAxisAlignment.center,
            children: <Widget>[
              new RaisedButton(
                onPressed: () async {
                  final ConfirmAction action = await _asyncConfirmDial
                  print("Confirm Action $action" );
                },
                child: const Text(
                  "Show Alert",
                  style: TextStyle(fontSize: 20.0),),
                  padding: EdgeInsets.fromLTRB(30.0,10.0,30.0,10.0),
                  shape: RoundedRectangleBorder(
                    borderRadius: BorderRadius.circular(8.0)
                  ),
                  color: Colors.green,
                ),
            ],
          ),
        ),
      );
    }
  }
  enum ConfirmAction { Cancel, Accept}
  Future<ConfirmAction> _asyncConfirmDialog(BuildContext context) asyn
    return showDialog<ConfirmAction>(
      context: context,
      barrierDismissible: false, // user must tap button for close dia
      builder: (BuildContext context) {
        return AlertDialog(
          title: Text('Delete This Contact?'),
          content: const Text(
              'This will delete the contact from your device.'),
          actions: <Widget>[
            FlatButton(
              child: const Text('Cancel'),
              onPressed: () {
                Navigator.of(context).pop(ConfirmAction.Cancel);
              },
            ),
            FlatButton(
```

```
              child: const Text('Delete'),
              onPressed: () {
                Navigator.of(context).pop(ConfirmAction.Accept);
              },
            )
          ],
        );
      },
    );
  }
```

Đầu ra

Khi bạn chạy ứng dụng, nó sẽ đưa ra kết quả sau. Bây giờ, bấm vào nút Show Alert, bạn sẽ nhận được thông báo hộp cảnh báo xác nhận.



Chọn Option AlertDialog

Loại hộp thoại cảnh báo này hiển thị danh sách các mục, sẽ có hành động ngay lập tức khi được chọn.

Thí dụ

```
import 'package:flutter/material.dart';

void main() {
  runApp(new MaterialApp(home: new MyApp()));
}

class MyApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    // TODO: implement build
    return new Scaffold(
      appBar: AppBar(
```

```
            title: Text("Select Option AlertDialog"),
          ),
          body: Center(
            child: Column(
              mainAxisAlignment: MainAxisAlignment.center,
              children: <Widget>[
                new RaisedButton(
                  onPressed: () async {
                    final Product prodName = await _asyncSimpleDialog(co
                    print("Selected Product is $prodName");
                  },
                  child: const Text(
                    "Show Alert",
                    style: TextStyle(fontSize: 20.0),),
                    padding: EdgeInsets.fromLTRB(30.0,10.0,30.0,10.0),
                    shape: RoundedRectangleBorder(
                      borderRadius: BorderRadius.circular(8.0)
                    ),
                    color: Colors.green,
                  ),
              ],
            ),
          ),
        );
    }
  }
  enum Product { Apple, Samsung, Oppo, Redmi }

  Future<Product> _asyncSimpleDialog(BuildContext context) async {
    return await showDialog<Product>(
        context: context,
        barrierDismissible: true,
        builder: (BuildContext context) {
          return SimpleDialog(
            title: const Text('Select Product '),
            children: <Widget>[
              SimpleDialogOption(
                onPressed: () {
                  Navigator.pop(context, Product.Apple);
                },
                child: const Text('Apple'),
              ),
              SimpleDialogOption(
                onPressed: () {
                  Navigator.pop(context, Product.Samsung);
```

```
                },
                child: const Text('Samsung'),
              ),
              SimpleDialogOption(
                onPressed: () {
                  Navigator.pop(context, Product.Oppo);
                },
                child: const Text('Oppo'),
              ),
              SimpleDialogOption(
                onPressed: () {
                  Navigator.pop(context, Product.Redmi);
                },
                child: const Text('Redmi'),
              ),
            ],
          );
        });
  }
```

Đầu ra

Khi bạn chạy ứng dụng, nó sẽ đưa ra kết quả sau. Bây giờ, bấm vào nút Hiển thị cảnh báo, bạn sẽ nhận được thông báo hộp cảnh báo tùy chọn lựa chọn. Ngay sau khi bạn chọn bất kỳ tùy chọn có sẵn nào, thông báo cảnh báo sẽ biến mất và bạn sẽ nhận được thông báo về lựa chọn đã chọn trong bảng điều khiển.

Cài ứng dụng cafedev để dễ dàng cập nhật tin và học lập trình mọi lúc mọi nơi tại đây.

Tài liệu từ cafedev:

- Full series tự học Flutter từ cơ bản tới nâng cao tại đây nha.
- **Các nguồn kiến thức MIỄN PHÍ VÔ GIÁ từ cafedev tại đây**

Nếu bạn thấy hay và hữu ích, bạn có thể tham gia các kênh sau của cafedev để nhận được nhiều hơn nữa:

- **Group Facebook**
- **Fanpage**
- **Youtube**
- **Instagram**
- **Twitter**

- **Linkedin**
- **Pinterest**
- **Trang chủ**

Chào thân ái và quyết thắng!

Đăng ký kênh youtube để ủng hộ Cafedev nha các bạn, Thanks you!

Bài trước

Tự học Flutter | Tìm hiểu về các widget về Forms trong Flutter

Bài tiếp theo

Tự học Flutter | Tìm hiểu về widget Icon trong Flutter

David Xuân

https://cafedev.vn/

Thế giới ngôn ngữ lập trình - Programming Languages | Tự học Flutter

Tự học Flutter | Tìm hiểu về widget Row và Column trong Flutter

Flutter Row and Column

Bởi **David Xuân** - 21 Tháng Tư, 2021 997 0

Trong các phần trước, chúng ta đã học cách tạo một ứng dụng Flutter đơn giản và cách tạo kiểu cơ bản của nó cho các widget. Bây giờ, chúng ta sẽ học cách **sắp xếp các widget theo hàng và cột trên màn hình** . Các hàng và cột không phải là một widget, chúng là hai widget khác nhau, cụ thể là Hàng(Row) và Cột(Column). Ở đây, chúng ta sẽ tích hợp hai widget này với nhau vì chúng có các thuộc tính tương tự nhau giúp chúng ta hiểu chúng một cách hiệu quả và nhanh chóng.

Hàng và cột là hai widget thiết yếu trong Flutter cho phép các nhà phát triển **căn chỉnh widget con theo chiều ngang và chiều dọc theo nhu cầu của chúng ta** . Các widget này rất cần thiết khi chúng ta thiết kế giao diện người dùng ứng dụng trong Flutter.

Nội dung chính

1. Những điểm chính

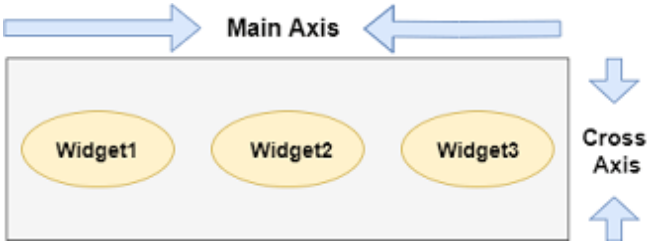
1. Các widget Hàng và Cột là các mẫu bố cục được sử dụng phổ biến nhất trong ứng dụng Flutter.
2. Cả hai đều có thể giữ một số widget con.
3. Một widget con cũng có thể là một widget hàng hoặc cột.
4. Chúng tôi có thể kéo dài hoặc hạn chế một widget cụ thể dành cho các widget con.
5. Flutter cũng cho phép các nhà phát triển chỉ định cách các widget con có thể sử dụng không gian có sẵn của các widget hàng và cột.

2. widget row

Widget này sắp xếp các con của nó theo hướng nằm ngang trên màn hình . Nói cách khác, nó sẽ mong đợi các widget con trong một mảng ngang. Nếu widget cần lấp đầy không gian ngang có sẵn, chúng ta phải bọc các widget trong widget Mở rộng.

widget hàng có vẻ **không cuộn được** vì nó hiển thị các widget trong chế độ xem và hiển thị toàn bộ. Vì vậy, sẽ được coi là sai nếu chúng ta có nhiều widget cob hơn trong một hàng sẽ không phù hợp với không gian có sẵn. Nếu chúng ta muốn tạo một danh sách các widget hàng có thể cuộn được, chúng ta cần sử dụng widget ListView.

Chúng ta có thể kiểm soát cách một widget hàng căn chỉnh các con của nó dựa trên lựa chọn của chúng tôi bằng cách sử dụng thuộc tính **crossAxisAlignment** và **mainAxisAlignment** . **Trục chéo** của hàng sẽ chạy **theo chiều dọc** và **trục chính** sẽ chạy **theo chiều ngang** . Xem hình ảnh minh họa bên dưới để hiểu rõ hơn.



Lưu ý: widget hàng Flutter có một số thuộc tính khác như mainAxisSize, textDirection, verticalDirection, v.v. Ở đây, chúng ta sẽ chỉ thảo luận về các thuộc tính mainAxisAlignment và crossAxisAlignment.

Chúng ta có thể căn chỉnh widget của hàng với sự trợ giúp của các thuộc tính sau:

- **start:** Nó sẽ đặt các con từ điểm bắt đầu của trục chính.

- **end:** Nó sẽ đặt các con ở cuối trục chính.
- **center:** Nó sẽ đặt các con ở giữa trục chính.
- **spaceBetween:** Nó sẽ đặt không gian trống giữa các con một cách đồng đều.
- **spaceAround:** Nó sẽ đặt không gian trống giữa các con một cách đồng đều và một nửa không gian đó trước và sau widget con đầu tiên và cuối cùng.
- **spaceEvenly:** Nó sẽ đặt không gian trống giữa các con một cách đồng đều và trước và sau widget con đầu tiên và cuối cùng.

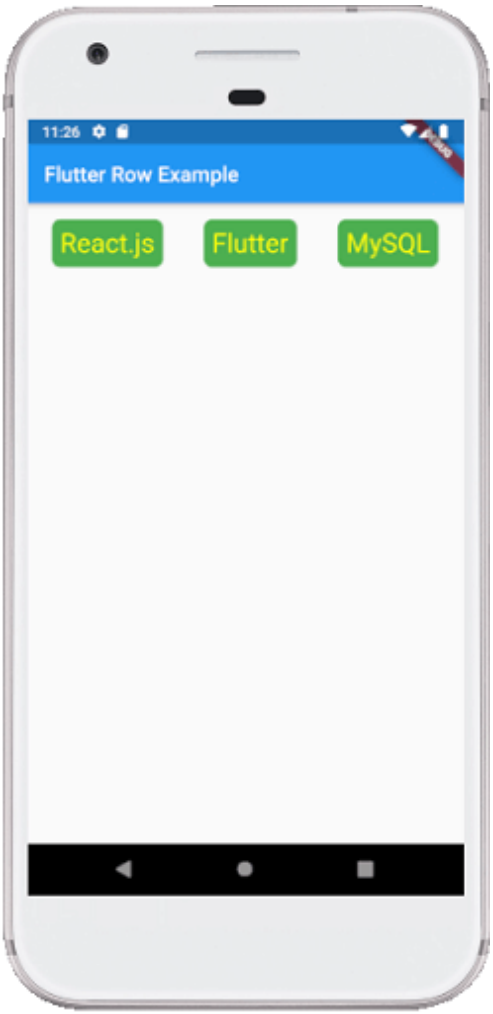
Hãy để chúng ta hiểu điều đó với sự trợ giúp của một ví dụ mà chúng ta sẽ sắp xếp nội dung sao cho có khoảng cách đồng đều xung quanh các con trong một hàng:

```
import 'package:flutter/material.dart';

void main() { runApp(MyApp()); }
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: MyHomePage()
    );
  }
}

class MyHomePage extends StatefulWidget {
  @override
  _MyHomePageState createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("Flutter Row Example"),
      ),
      body: Row(
          mainAxisAlignment: MainAxisAlignment.spaceEvenly,
          children:<Widget>[
            Container(
              margin: EdgeInsets.all(12.0),
              padding: EdgeInsets.all(8.0),
```

```
                decoration:BoxDecoration(
                    borderRadius:BorderRadius.circular(8),
                    color:Colors.green
                ),
                child: Text("React.js",style: TextStyle(color:Colors.y
              ),
              Container(
                margin: EdgeInsets.all(15.0),
                padding: EdgeInsets.all(8.0),
                decoration:BoxDecoration(
                    borderRadius:BorderRadius.circular(8),
                    color:Colors.green
                ),
                child: Text("Flutter",style: TextStyle(color:Colors.ye
              ),
              Container(
                margin: EdgeInsets.all(12.0),
                padding: EdgeInsets.all(8.0),
                decoration:BoxDecoration(
                    borderRadius:BorderRadius.circular(8),
                    color:Colors.green
                ),
                child: Text("MySQL",style: TextStyle(color:Colors.yell
              )
            ]
        ),
      );
    }
  }
```

Đầu ra:

Khi chúng ta chạy ứng dụng này, chúng tôi sẽ nhận được giao diện người dùng như ảnh chụp màn hình bên dưới.

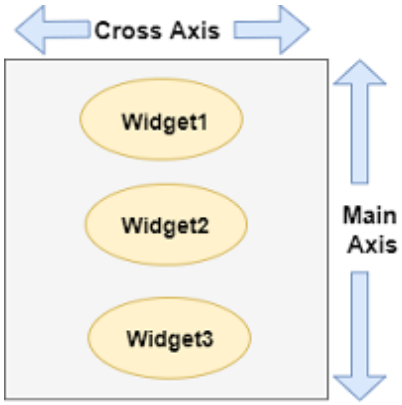


3.widget column

Widget này sắp xếp các con của nó theo hướng dọc trên màn hình . Nói cách khác, nó sẽ mong đợi một mảng dọc gồm các widget con. Nếu widget cần lấp đầy không gian dọc có sẵn, chúng ta phải bọc các widget trong widget Mở rộng.

Một widget dạng cột **không thể cuộn được** vì nó hiển thị các widget trong chế độ xem hiển thị đầy đủ. Vì vậy, sẽ được coi là sai nếu chúng ta có nhiều con hơn trong một cột sẽ không phù hợp với không gian có sẵn. Nếu chúng ta muốn tạo một danh sách các widget cột có thể cuộn được, chúng ta cần sử dụng ListView Widget.

Chúng ta cũng có thể kiểm soát cách một widget cột sắp xếp các con của nó bằng cách sử dụng thuộc tính mainAxisAlignment và crossAxisAlignment. **Trục chéo** của cột sẽ chạy **theo chiều ngang** và **trục chính** sẽ chạy **theo chiều dọc** . Hình ảnh minh họa dưới đây giải thích rõ ràng hơn.



Lưu ý: widget cột cũng căn chỉnh nội dung của nó bằng cách sử dụng các thuộc tính tương tự như chúng ta đã thảo luận trong widget hàng như bắt đầu, kết thúc, giữa, khoảng trắng, khoảng trắng và khoảng trắng.

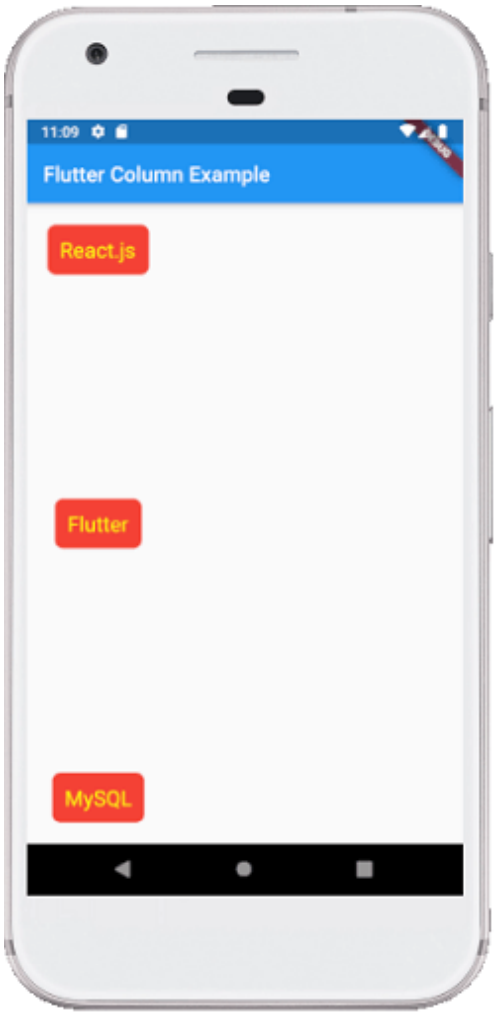
Hãy để chúng ta hiểu điều đó với sự trợ giúp của một ví dụ trong đó chúng ta sẽ căn chỉnh nội dung sao cho có một khoảng trống giữa các phần tử con đều nhau trong một cột:

```
import 'package:flutter/material.dart';

void main() { runApp(MyApp()); }
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
        home: MyHomePage()
    );
  }
}

class MyHomePage extends StatefulWidget {
  @override
  _MyHomePageState createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("Flutter Column Example"),
      ),
      body: Column(
```

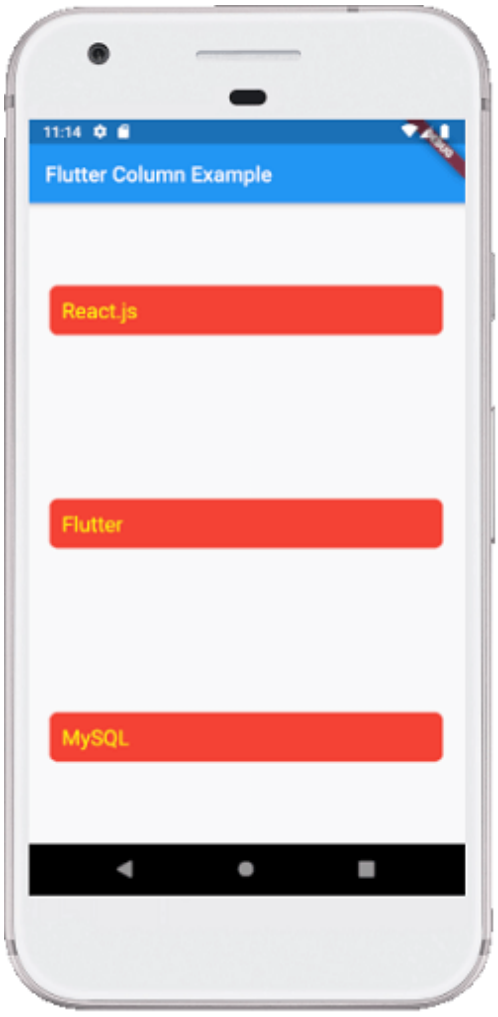
```
            mainAxisAlignment: MainAxisAlignment.spaceBetween,
            children:<Widget>[
              Container(
                margin: EdgeInsets.all(20.0),
                padding: EdgeInsets.all(12.0),
                decoration:BoxDecoration(
                    borderRadius:BorderRadius.circular(8),
                    color:Colors.red
                ),
                child: Text("React.js",style: TextStyle(color:Colors.y
              ),
              Container(
                margin: EdgeInsets.all(20.0),
                padding: EdgeInsets.all(12.0),
                decoration:BoxDecoration(
                    borderRadius:BorderRadius.circular(8),
                    color:Colors.red
                ),
                child: Text("Flutter",style: TextStyle(color:Colors.ye
              ),
              Container(
                margin: EdgeInsets.all(20.0),
                padding: EdgeInsets.all(12.0),
                decoration:BoxDecoration(
                    borderRadius:BorderRadius.circular(8),
                    color:Colors.red
                ),
                child: Text("MySQL",style: TextStyle(color:Colors.yell
              )
            ]
        ),
      );
  }
}
```

Đầu ra:

Khi chúng ta chạy ứng dụng này, chúng tôi sẽ nhận được giao diện người dùng như ảnh chụp màn hình bên dưới.



Flutter cũng cho phép các nhà phát triển căn chỉnh widget với sự kết hợp của crossAxisAlignment và mainAxisAlignment cho cả widget hàng và cột . Chúng ta hãy lấy ví dụ ở trên về widget cột, nơi chúng ta sẽ đặt mainAxisAlignment là MainAxisAlignment.spaceAround và crossAxisAlignment là CrossAxisAlignment.stretch. Nó sẽ làm cho chiều cao của cột bằng với chiều cao của cơ thể. Xem ảnh chụp màn hình bên dưới.



4. Hạn chế của widget row và column:

- widget row trong Flutter không có tính năng cuộn theo chiều ngang. Vì vậy, nếu chúng ta đã chèn một số lượng lớn con trong một hàng mà không thể vừa với hàng đó, chúng ta sẽ thấy thông báo Overflow.
- widget column trong Flutter không có tính năng cuộn dọc. Vì vậy, nếu chúng tôi đã chèn một số lượng lớn widget con trong một cột duy nhất có tổng kích thước widget con không bằng chiều cao của màn hình, chúng ta sẽ thấy thông báo Overflow.

Cài ứng dụng cafedev để dễ dàng cập nhật tin và học lập trình mọi lúc mọi nơi tại đây.

Tài liệu từ cafedev:

- Full series tự học Flutter từ cơ bản tới nâng cao tại đây nha.
- **Các nguồn kiến thức MIỄN PHÍ VÔ GIÁ từ cafedev tại đây**

Nếu bạn thấy hay và hữu ích, bạn có thể tham gia các kênh sau của cafedev để nhận được nhiều hơn nữa:

- **Group Facebook**
- **Fanpage**
- **Youtube**
- **Instagram**
- **Twitter**
- **Linkedin**
- **Pinterest**
- **Trang chủ**

Chào thân ái và quyết thắng!

Đăng ký kênh youtube để ủng hộ Cafedev nha các bạn, Thanks you!

Bài trước

Tự học Flutter | Tìm hiểu về widget Container trong Flutter

Bài tiếp theo

Tự học Flutter | Tìm hiểu về widget Text trong Flutter

David Xuân

https://cafedev.vn/

Tự học Flutter | Tìm hiểu về các widget về Forms trong Flutter

Flutter Forms

Bởi **David Xuân** - 21 Tháng Tư, 2021 789 0

Biểu mẫu(Forms) là một phần không thể thiếu của tất cả các ứng dụng web và di động hiện đại. Nó chủ yếu được sử dụng để tương tác với ứng dụng cũng như thu thập thông tin từ người dùng. Họ có thể thực hiện nhiều tác vụ, tùy thuộc vào bản chất của yêu cầu và logic kinh doanh của bạn, chẳng hạn như xác thực người dùng, thêm người dùng, tìm kiếm, lọc, đặt hàng, đặt chỗ, v.v. Một biểu mẫu có thể chứa các trường văn bản, nút, hộp kiểm, radio các nút, v.v.

Nội dung chính

1. Tạo biểu mẫu

Flutter cung cấp **widget Biểu mẫu** để tạo biểu mẫu. widget biểu mẫu hoạt động như một vùng chứa, cho phép chúng ta nhóm và xác thực nhiều trường biểu mẫu. Khi bạn tạo biểu mẫu, bạn cần cung cấp **GlobalKey** . Khóa này xác định duy nhất biểu mẫu và cho phép bạn thực hiện bất kỳ xác thực nào trong các trường biểu mẫu.

widget biểu mẫu sử dụng widget con **TextFormField** để cung cấp cho người dùng nhập trường văn bản. widget này hiển thị trường văn bản material design và cũng cho phép chúng ta hiển thị các lỗi xác thực khi chúng xảy ra.

Hãy để chúng ta tạo một biểu mẫu. Đầu tiên, tạo một dự án Flutter và thay thế code sau trong tệp main.dart. Trong đoạn code này, chúng ta đã tạo một lớp tùy chỉnh có tên **MyCustomForm** . Bên trong lớp này, chúng ta định nghĩa một khóa toàn cục là **_formKey** . Khóa này giữ một **FormState** và có thể sử dụng để truy xuất widget biểu mẫu. Bên trong phương thức **xây dựng** của lớp này, chúng ta đã thêm một số kiểu tùy chỉnh và sử dụng widget TextFormField để cung cấp các trường biểu mẫu như tên, số điện thoại, ngày sinh hoặc chỉ một trường bình thường. Bên trong TextFormField, chúng ta đã sử dụng **InputDecoration** để cung cấp giao diện của các thuộc tính biểu mẫu của bạn như đường viền, nhãn, biểu tượng, gợi ý, kiểu, v.v. Cuối cùng, chúng ta đã thêm một **nút** để gửi biểu mẫu.

```
import 'package:flutter/material.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final appTitle = 'Flutter Form Demo';
    return MaterialApp(
      title: appTitle,
      home: Scaffold(
        appBar: AppBar(
          title: Text(appTitle),
        ),
        body: MyCustomForm(),
      ),
    );
  }
}
// Create a Form widget.
class MyCustomForm extends StatefulWidget {
  @override
  MyCustomFormState createState() {
    return MyCustomFormState();
  }
}
// Create a corresponding State class. This class holds data related
```

```dart
class MyCustomFormState extends State<MyCustomForm> {
  // Create a global key that uniquely identifies the Form widget
  // and allows validation of the form.
  final _formKey = GlobalKey<FormState>();

  @override
  Widget build(BuildContext context) {
    // Build a Form widget using the _formKey created above.
    return Form(
      key: _formKey,
      child: Column(
        crossAxisAlignment: CrossAxisAlignment.start,
        children: <Widget>[
          TextFormField(
            decoration: const InputDecoration(
              icon: const Icon(Icons.person),
              hintText: 'Enter your name',
              labelText: 'Name',
            ),
          ),
          TextFormField(
            decoration: const InputDecoration(
              icon: const Icon(Icons.phone),
              hintText: 'Enter a phone number',
              labelText: 'Phone',
            ),
          ),
          TextFormField(
            decoration: const InputDecoration(
            icon: const Icon(Icons.calendar_today),
            hintText: 'Enter your date of birth',
            labelText: 'Dob',
            ),
           ),
          new Container(
              padding: const EdgeInsets.only(left: 150.0, top: 40.0),
              child: new RaisedButton(
                child: const Text('Submit'),
                  onPressed: null,
              )),
        ],
      ),
    );
  }
}
```

Đầu ra

Bây giờ, hãy chạy ứng dụng, bạn có thể thấy màn hình sau trong Trình giả lập Android của mình. Biểu mẫu này có ba tên trường, số điện thoại, ngày sinh và nút gửi.

2. Biểu mẫu có validation

Xác thực(validation) là một phương pháp, cho phép chúng ta sửa chữa hoặc xác nhận một tiêu chuẩn nhất định. Nó đảm bảo xác thực dữ liệu đã nhập.

Xác thực biểu mẫu là một thực tế phổ biến trong tất cả các tương tác kỹ thuật số. Để xác thực một biểu mẫu trong nháy mắt, chúng ta cần thực hiện chủ yếu ba bước.

Bước 1: Sử dụng widget Biểu mẫu với khóa chung.

Bước 2: Sử dụng TextFormField để cung cấp cho trường đầu vào thuộc tính trình xác thực.

Bước 3: Tạo nút để xác thực các trường biểu mẫu và hiển thị lỗi xác thực.

Hãy để chúng ta hiểu nó với ví dụ sau. Trong đoạn code trên, chúng ta phải sử dụng hàm **validator()** trong TextFormField để xác thực các thuộc tính đầu vào. Nếu người dùng nhập sai, hàm xác nhận sẽ trả về một chuỗi có chứa thông **báo lỗi** ; nếu không, hàm xác nhận sẽ trả về **null** . Trong hàm trình xác thực, hãy đảm bảo rằng TextFormField không trống. Nếu không, nó sẽ trả về một thông báo lỗi.

Hàm validator() có thể được viết dưới dạng đoạn code dưới đây:

```
validator: (value) {
        if (value.isEmpty) {
              return 'Please enter some text';
        }
        return null;
}
```

Bây giờ, mở tệp **main.dart** và thêm hàm validator() trong widget TextFormField. Thay thế code sau bằng tệp main.dart.

```
import 'package:flutter/material.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final appTitle = 'Flutter Form Demo';
    return MaterialApp(
      title: appTitle,
      home: Scaffold(
        appBar: AppBar(
          title: Text(appTitle),
        ),
        body: MyCustomForm(),
      ),
    );
  }
}
// Create a Form widget.
class MyCustomForm extends StatefulWidget {
```

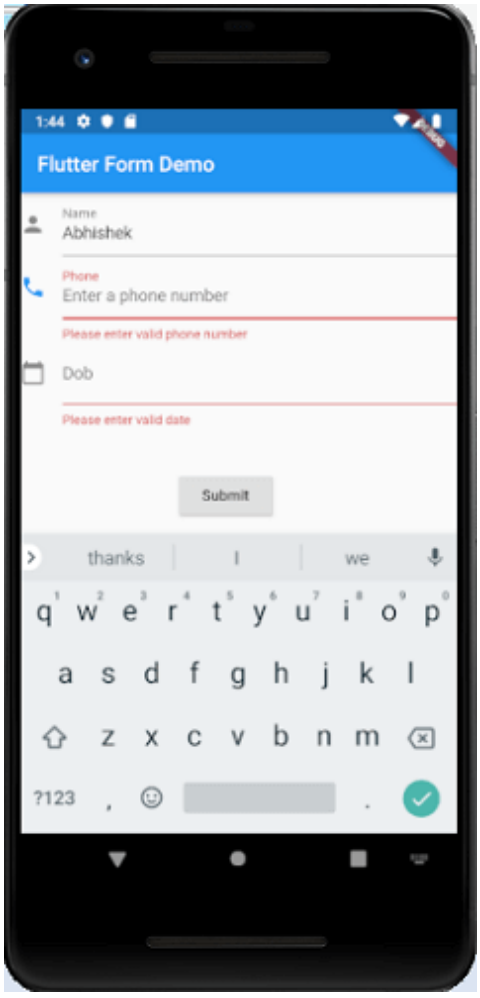
```
  @override
  MyCustomFormState createState() {
    return MyCustomFormState();
  }
}
// Create a corresponding State class, which holds data related to t
class MyCustomFormState extends State<MyCustomForm> {
  // Create a global key that uniquely identifies the Form widget
  // and allows validation of the form.
  final _formKey = GlobalKey<FormState>();

  @override
  Widget build(BuildContext context) {
    // Build a Form widget using the _formKey created above.
    return Form(
      key: _formKey,
      child: Column(
        crossAxisAlignment: CrossAxisAlignment.start,
        children: <Widget>[
          TextFormField(
            decoration: const InputDecoration(
              icon: const Icon(Icons.person),
              hintText: 'Enter your full name',
              labelText: 'Name',
            ),
            validator: (value) {
              if (value.isEmpty) {
                return 'Please enter some text';
              }
              return null;
            },
          ),
          TextFormField(
            decoration: const InputDecoration(
              icon: const Icon(Icons.phone),
              hintText: 'Enter a phone number',
              labelText: 'Phone',
            ),
            validator: (value) {
              if (value.isEmpty) {
                return 'Please enter valid phone number';
              }
              return null;
            },
          ),
```

```
            TextFormField(
              decoration: const InputDecoration(
              icon: const Icon(Icons.calendar_today),
              hintText: 'Enter your date of birth',
              labelText: 'Dob',
              ),
              validator: (value) {
                if (value.isEmpty) {
                  return 'Please enter valid date';
                }
                return null;
              },
             ),
            new Container(
                padding: const EdgeInsets.only(left: 150.0, top: 40.0)
                child: new RaisedButton(
                  child: const Text('Submit'),
                  onPressed: () {
                    // It returns true if the form is valid, otherwise
                    if (_formKey.currentState.validate()) {
                      // If the form is valid, display a Snackbar.
                      Scaffold.of(context)
                          .showSnackBar(SnackBar(content: Text('Data i
                    }
                  },
                )),
          ],
        ),
      );
  }
}
```

Đầu ra

Bây giờ, hãy chạy ứng dụng. Màn hình sau xuất hiện.

Trong biểu mẫu này, nếu bạn để trống bất kỳ trường nhập liệu nào, bạn sẽ nhận được thông báo lỗi như màn hình dưới đây.



Cài ứng dụng cafedev để dễ dàng cập nhật tin và học lập trình mọi lúc mọi nơi tại đây.

Tài liệu từ cafedev:

- Full series tự học Flutter từ cơ bản tới nâng cao tại đây nha.
- **Các nguồn kiến thức MIỄN PHÍ VÔ GIÁ từ cafedev tại đây**

Nếu bạn thấy hay và hữu ích, bạn có thể tham gia các kênh sau của cafedev để nhận được nhiều hơn nữa:

- **Group Facebook**
- **Fanpage**
- **Youtube**
- **Instagram**
- **Twitter**

- **Linkedin**
- **Pinterest**
- **Trang chủ**

Chào thân ái và quyết thắng!

Đăng ký kênh youtube để ủng hộ Cafedev nha các bạn, Thanks you!

Bài trước

Tự học Flutter | Tìm hiểu về widget Stack trong Flutter

Bài tiếp theo

Tự học Flutter | Tìm hiểu về widget Alert Dialogs trong Flutter

David Xuân

https://cafedev.vn/

```
Widget build(BuildContext context) {
return Scaffold(
    appBar: AppBar(
    title: Text('First Flutter Application'),
    ),
    body: Center(
    child: Text("Welcome to Cafedev.vn",
        style: TextStyle( color: Colors.black, fontSize: 30.0,
        ),
         ),
    ),
}
```